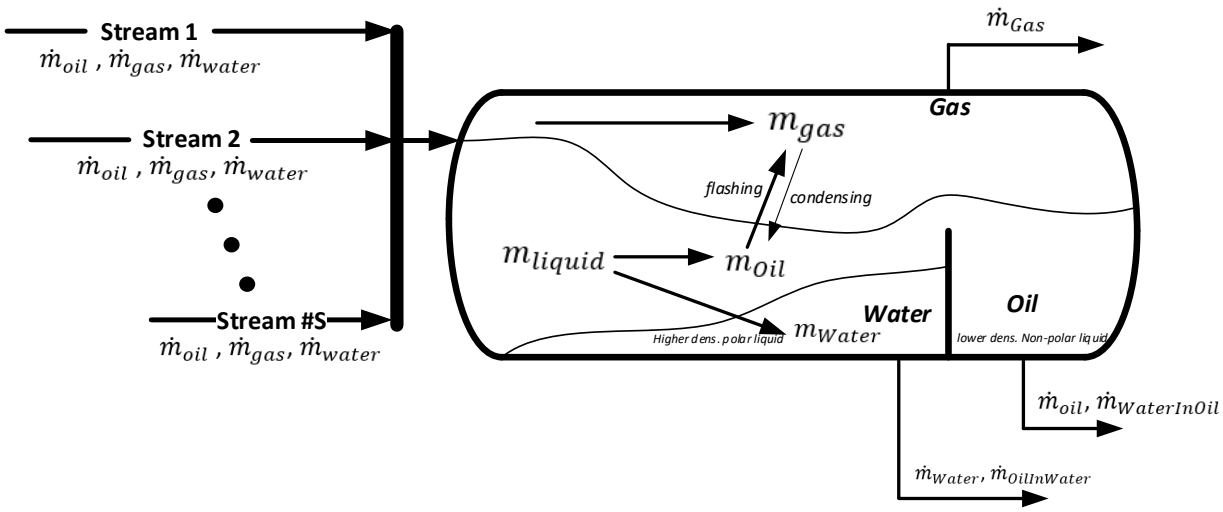
University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2019

Industrial IT and Automation

Parallel calibration of multiphase flow meters (MPFM) based on measurements of phase streams in separators



$$For\ p = [oil, gas, water]$$

$$\begin{bmatrix} \int_{t_{0,1}}^{t_{1,1}} \frac{dm_{p,1,1}}{dt} dt & \int_{t_{0,1}}^{t_{1,1}} \frac{dm_{p,2,1}}{dt} dt & \cdots & \int_{t_{0,1}}^{t_{1,1}} \frac{dm_{p,S,1}}{dt} dt \\ \int_{t_{0,2}}^{t_{1,2}} \frac{dm_{p,1,2}}{dt} dt & \int_{t_{0,2}}^{t_{1,2}} \frac{dm_{p,2,2}}{dt} dt & \cdots & \int_{t_{0,2}}^{t_{1,2}} \frac{dm_{p,S,2}}{dt} dt \\ \vdots & \vdots & \ddots & \vdots \\ \int_{t_{0,S}}^{t_{1,S}} \frac{dm_{p,1,3}}{dt} dt & \int_{t_{0,S}}^{t_{1,S}} \frac{dm_{p,2,3}}{dt} dt & \cdots & \int_{t_{0,S}}^{t_{1,S}} \frac{dm_{p,S,T}}{dt} dt \end{bmatrix} \cdot \begin{bmatrix} k_{p,1} \\ k_{p,2} \\ \vdots \\ k_{p,S} \end{bmatrix} = \begin{bmatrix} \int_{t_{0,1}}^{t_{1,1}} \frac{dm_{p,ref,1}}{dt} dt \\ \int_{t_{0,2}}^{t_{1,2}} \frac{dm_{p,ref,2}}{dt} dt \\ \vdots \\ \int_{t_{0,S}}^{t_{1,S}} \frac{dm_{p,ref,T}}{dt} dt \end{bmatrix}$$

Stig Harald Gustavsen

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis, 2019

Title: Parallel calibration of multiphase flow meters (MPFM) based on measurements of phase streams in separators

Number of pages: 74

Keywords: Parallel Calibration, Cognite Data Fusion, Multiphase flow meters, Fiscal Oil and Gas Metering and Allocation, Scientific computing, Computational Engineering, Python, Data Fusion, Digital Twin, Object Oriented Data Science, Condition Based Maintenance

Student: Stig Harald Gustavsen

Supervisor: Saba Mylvaganam

External partner: Torbjørn Selanger

Availability: Open, with selected confidential appendices.

Summary:

The Alvheim field suffers from significant production deferrals of oil and gas, during calibration of multiphase flow meters used in ownership allocation. This thesis has developed an algorithm solving a new method, which effectively negates these deferrals. This is done through an object-oriented data science approach, in creating a framework for performing these calibrations in an elegant and efficient manner. The algorithm has been tested and compared to real world data and shows promising results. The tests during April 2019 showed an increase of 15000bbl of oil production during parallel calibration compared to a normal calibration. The Cognite Data Fusion repository helped in streamlining the development process with easy and swift access to process data. The algorithm was implemented and developed in the programming language Python. Additionally, this thesis covers the purpose and technical background of ownership allocation measurements and the systems and sensors involved in measurement and calibration. The details of the developed algorithm, and the calibration results are presented and discussed.

Preface

This master thesis is owing a huge amount of gratitude to all the numerical integrators doing summation operations of tiny increments or finite differences around the world. Both as a cumulative operator or numerically solving differential equations, giving important insight into how the laws of nature act and how we humans can use this for our own benefit, humanity owes a huge thanks for your tedious work.

This Thesis will develop an implementation of an algorithm which uses a method initially proposed by the process engineer in Torbjørn Selanger, where the idea came when a process technician named Peter Kongstad Schmidt asked "*why this can't be solved at the same time*", which peaked Torbjørn's thoughts and the idea of the method was created, this was then linearized by Therese Renstrøm. Both Torbjørn and Therese are working in the production optimization group of the digitalization program called Eureka in Aker BP ASA. And without anyone of these people this method would not have seen the light of day and no algorithm for me to develop.

The implementation was done in the programming language of python with the use of NumPy and Pandas libraries for data manipulation and calculations, and the Cognite Data Fusion repository and the data source through the Cognite Python SDK. The diagrams and drawings in this thesis were drawn with Microsoft Visio. It is suggested that the reader of this paper have some previous exposure to process technology, instrumentation, P&ID notation, and an understanding of fundamentals in programming, numerical mathematical methods and linear algebra.

Topic description, a work breakdown structure and a Gantt for the planned execution and baseline of this thesis is in appendix A.

I want to thank my teachers, professors and organizers at the University of South-Eastern Norway (USN), for enabling me to execute both an engineering bachelor and a Master of Science program, while working a fulltime job at Aker BP ASA these last 7 years. And further thank the Norwegian society for oil and gas measurement (NFOGM) and the MPM team in TechnipFMC for both the information and to be allowed to share some of their produced figures within my thesis. But not to mention the teams working with Python and its related frameworks and libraries; NumPy, SciPy, PyData's Pandas, Matplotlib, IPython and their spin off project Jupyter for enabling anyone to do scientific computing with great libraries, fantastic development environment, freely and open source. A last shout out to the production crew working onboard the Alvheim field in the North Sea, whom has spent several days rerouting fluids in the process system onboard, these last few months, providing real data used in this thesis.

I've been incredibly lucky with the thesis topic which are on subjects near and dear to my passions of; applications of numerical integrators, scientific computing and last but not least precise and representative measurements of oil and gas. Good luck to the reader, I hope this collection of documents are of interest, and that the figures are more appreciated than the text.

Porsgrunn / Valhall, 13/05-2019

Stig Harald Gustavsen

Contents

Preface 3

Contents 4

List of figures 7

Acronym 9

Symbols 10

1 Introduction 11

2 Petroleum industry in Norway 13

2.1 History 13
2.2 Licenses and Blocks 13
2.3 Ownership, production metering, allocation and hydrocarbon management 14
2.3.1 Allocation ***14***
2.3.2 Measurement and measurands for allocation of oil and gas. ***14***
2.3.3 Field Blend ***15***
2.4 Future prospects and focus on the Norwegian Continental Shelf (NCS) 16

3 Separation and flow of fluids - Brief theoretical background 17

3.1 Hydrocarbon fluid and separation 17
3.2 Fluid flow 17
3.2.1 Phase separation ***18***
3.2.2 Single-Phase flow stream ***18***
3.2.3 Multiphase flows stream ***18***
3.2.4 Accumulation of flow ***20***
3.3 Balance laws 21
3.4 Modeling of the dynamic phenomena 21
3.4.1 Separator modeling; balance laws and phase mass exchange dynamics ***21***
3.5 Flashing, PVT and Phase equilibrium 23

4 Technical background of hydrocarbon flow metering 24

4.1 Instrumentation 24
4.2 Liquid flow measurement 24
4.2.1 Oil flow measurement ***25***
4.2.2 Calibration and traceability of liquid volume flow meters ***26***
4.3 Gas flow measurements 28
4.4 The computer part – Flow computer 29
4.5 Multiphase flow meters 30
4.5.1 Density measurement ***31***
4.5.2 Velocity measurement of fluids ***31***
4.5.3 Tomographic measurement ***32***
4.6 Operations / Maintenance of measurement equipment and systems 33
4.7 Uncertainty 33
4.7.1 Traceability ***34***
4.8 Supervisory metering system 34
4.8.1 Data connectivity and interfaces. ***35***

5 Alvheim third party field installation 37

5.1 Alvheim field in general 37
5.2 Topside MPFM Manifold 37

Contents

5.3 Third Party Separator 38
5.3.1 Oil stream *39*
5.3.2 Gas Stream *41*

6 Parallel calibration of multiphase flow meters 43

6.1 Digital representation of fluid streams 43
6.2 Traditional calibration method 44
6.2.1 Calibration factor calculation *45*
6.2.2 Acceptance criteria for a traditional calibration *47*
6.3 Parallel calibration method 48
6.3.1 Calibration factor calculation *48*
6.3.2 Non-linear solver *49*
6.3.3 Linear solver *49*
6.4 Parallel calibration algorithm 51
6.4.1 Result and statistical analysis of the calibration method *52*
6.4.2 Calibration evaluation *53*
6.4.3 Synthesizing data for comparing traditional calibration towards a parallel calibration *54*
6.5 Executing parallel calibrations 54

7 Results 56

7.1 January calibration 56
7.2 April calibration results 59
7.3 Financial gains of using this algorithm 62

8 Discussion 63

8.1 Development 63
8.1.1 Non-available historical datapoints *63*
8.1.2 Historical logging of each flow computer increments *64*
8.1.3 Negligence of physical properties *64*
8.2 Parallel calibration 64
8.2.1 Trial quality, size and order *65*
8.3 Traditional vs Parallel calibration 65
8.4 The achieved result 65
8.5 Uncertainty 66
8.5.1 Traditional vs Synthetic parallel calibration *66*
8.5.2 Calibrating closer to normal operating conditions *66*
8.5.3 Numerical uncertainty in the parallel calibration method *67*
8.5.4 In calculation of flashing *67*
8.5.5 Accumulation quality *67*
8.6 Further work and development 67
8.6.1 Automatic multi trial combination *67*
8.6.2 Multi-dimensional multiphase meter calibration characteristic *67*
8.6.3 Increment database in flow computer to CDP *68*
8.6.4 Sliding time-window approach *68*
8.6.5 Visualizing the k-factor over runs *68*
8.6.6 Soft-sensor of multiphase meters *68*
8.6.7 Calculate separator streams to multiphase conditions *68*
8.6.8 Petrochemical calculations program development *68*
8.6.9 Digital twin giving further insight than planned *69*
8.7 Proposal for new structure for parallel calibration 70

9 Conclusion 71

References 72

Contents

Appendices **74**

List of figures

FIGURE 1 HYDROCARBON VALUE CHAIN 14
FIGURE 2 - PETROLEUM FIELD ALLOCATION BY MASS EXAMPLE ASSOCIATED WITH DIFFERENT LICENSE AGREEMENTS 16
FIGURE 3 - CONCEPT OF VOLUMETRIC FLOW AND DISCRETE ACCUMULATION, WHERE IS *U* VELOCITY AND u AVERAGE VELOCITY [13] 18
FIGURE 4 – MULTIPHASE FLOW WITH INCREASING GVF, WHERE THE BLUE PART IS LIQUID AND THE YELLOW REPRESENTS GAS AND GAS BUBBLES [15] 19
FIGURE 5 - TWO-PHASE FLOW MAP OF A VERTICAL PIPE [14] 20
FIGURE 6 - BLOCK SCHEMATIC ABSTRACTION OF MASS BALANCE 22
FIGURE 7 - MASS BALANCE OF MULTIPHASE STREAMS AND SEPARATOR 22
FIGURE 8 – LIQUID TURBINE METER FLOW CALIBRATION WITH COMPACT PROVER 27
FIGURE 9 - OVERVIEW OF TYPICAL AND SIMPLIFIED SINGLE-PHASE FLUID MEASUREMENT STREAM WITH THE ABSTRACT TASKS AND CALCULATIONS OF THE FLOW COMPUTER 29
FIGURE 10 - CONCEPTUAL OVERVIEW OF INSTRUMENTS INVOLVED IN A MULTIPHASE METER 30
FIGURE 11 MPM METER COMPONENTS [15] 31
FIGURE 12 - VENTURI CONE ELEMENT IN A MPM METER [15] 32
FIGURE 13 – TECHNIP FMC- MPM 3D BROADBAND™ TECHNOLOGY [15] 33
FIGURE 14 - TRACEABILITY MAP OF A TURBINE FLOW METER 34
FIGURE 15 - DATA FLOW FOR DATA USED IN THESIS 36
FIGURE 16 - ALVHEIM THIRD PARTY PRODUCTION AND MPM'S ON METERING SCADA 37
FIGURE 17 - VILJE AND VOLUND MPM - MPFMS 38
FIGURE 18 - ALVHEIM 3RD PARTY SEPARATOR 39
FIGURE 19 - SIMPLIFIED P&ID OVER THIRD-PARTY SEPARATORS OIL METERING STATION 40
FIGURE 20 - PICTURE OF OIL STREAM 1 WITH EXPLANATIONS 40
FIGURE 21 - OIL STREAM FAST-LOOP 41
FIGURE 22 - SIMPLIFIED P&ID OVER THIRD-PARTY SEPARATORS GAS METERING STREAMS 42
FIGURE 23 - CONCEPT OF DIGITAL TWIN OF SEPARATOR STREAMS 43
FIGURE 24 - CONCEPT OF DIGITAL TWIN OF A MULTIPHASE STREAM 44
FIGURE 25 - CONCEPTUAL OVERVIEW OF TRADITIONAL CALIBRATION METHOD 46
FIGURE 26 - NOT ACCEPTABLE K-FACTOR DEVELOPMENT THROUGH TRADITIONAL CALIBRATION ON THE METERING SYSTEM ON ALVHEIM 47
FIGURE 27 – ACCEPTABLE K-FACTOR DEVELOPMENT THROUGH TRADITIONAL CALIBRATION ON THE METERING SYSTEM ON ALVHEIM 48
FIGURE 28 - DATA BASIS, FLOW, CALCULATIONS, AND PREPARATION FOR PARALLEL CALIBRATION METHOD SOLVER 49
FIGURE 29 - INSIDE THE ALGORITHM - FILLING $\boldsymbol{Mp}$ MATRIX AND $\boldsymbol{mp}$ VECTOR WITH A 2X2X2 TRIAL INPUT 52
FIGURE 30 - EXAMPLE OF DETECTED STATISTICAL BASIS (RED) FROM K-FACTOR DEVELOPMENT (GRAY), VERTICAL SAMPLEL HISTOGRAM (GREEN) WITH A NORMAL DISTROBUTION PROBABILITY DENSITYPLOT (ORANGE) 53
FIGURE 31 - AUGMENTED MATRIX PLOT OF PROCESS CONDITIONS DURING TRIALS, WHERE THE X AXIS IS SUCCESSIVE RAW DATAPOINT DURING THE TRIAL, AND THEREFORE NO NUMBERS. 54
FIGURE 32 - FLOW CHART AND METHOD EXECUTION FOR A 3X2X2 CALIBRATION IN AN IPYTHON ENVIRONMENT 55
FIGURE 33 - K-FACTOR DEVELOPMENT OF TRADITIONAL VS SYNTHETIC PARALLEL CALIBRATION FOR BØYLA IN JANUARY 2019 56
FIGURE 34 - K-FACTOR DEVELOPMENT OF TRADITIONAL VS SYNTHETIC PARALLEL CALIBRATION FOR VILJE IN JANUARY 2019 57
FIGURE 35- K-FACTOR DEVELOPMENT OF TRADITIONAL VS SYNTHETIC PARALLEL CALIBRATION FOR VOLUND IN JANUARY 2019 57

List of figures

FIGURE 36 - RESULT COMPARISON OF VOLUND FROM JANUARY CALIBRATION TRIALS, WHERE THE REAL VALUE, TRADITIONAL AND A SYNTHETIC TEST WAS PERFORMED 59
FIGURE 37 - RESULT COMPARISON OF BØYLA DURING MARCH/APRIL CALIBRATIONS 60
FIGURE 38 - RESULT COMPARISON OF VOLUND DURING MARCH/APRIL CALIBRATIONS 61
FIGURE 39 - RESULT COMPARISON OF VILJE DURING MARCH/APRIL CALIBRATIONS 61
FIGURE 40 – QUALITATIVE ASPECTS OF SEPARATOR CAPACITY UTILIZATION UNDER PARALLEL AND TRADITIONAL CALIBRATION 62
FIGURE 41 - NEW STRUCTURE FOR PARALLEL CALIBRATION CONCEPT 70

Acronym

Acronym	Explanation
AGA	American Gas Association
API	American Petroleum Institute
ASA	Notation of a publicly listed company at stock exchange
BOE	Barrel of oil equivalent
CDF	Cognite Data Fusion – contextualized industrial data repository form Cognite
ERP	Enterprise resource and planning – abstract layer for the use of process information
E&P	Exploration & Production
FE	Flow Element
FPSO	Floating production storage and offloading
FQI	Flow Quantity Indicator (Accumulator in P&ID notation)
FT	Flow Transmitter
FWA	Flow Weighted Average
GOR	Gas oil ratio
GUM	Guide to the expression of uncertainty in measurement
GVF	Gas Volume Fraction
ISO	International Organization for Standardization
JCGM	Joint Committee for Guides in Metrology
MPE / OED	Ministry for Petroleum and energy
MPFM	Multi-phase Flow Meter
NCS	Norwegian continental shelf
NFOGM	Norwegian society for oil and gas measurement
NPD / OD	Norwegian petroleum directorate / Oljedirektoretatet
OIW	Oil In Water
OPC	Open Process control
P&ID	Piping and instrument diagram
PSA / Ptil	Petroleum safety Authority / Petroleumstilsynet
PVT	Pressure Volume Temperature with regards to Equations of state (correction and flashing)
SCADA	Supervisory Control And Data Acquisition
USFM	Ultrasonic flow meter
VOS	Velocity of sound
wgt%	weight percent fraction

Symbols

Symbol	Explanation	Unit
$\boldsymbol{k}_p$	Vector of mass-based k-factor for a specific phase-p $\in \mathbb{R}^S$	$[-]$
$\frac{dm_{p,s,T}}{dt}$	Mass-flowrate Leibniz notation, subscript; p-phase, s-stream, T-trial	$\frac{kg}{h}$
$\dot{m}_{p,s,T}$	Mass-flowrate Newtons notation subscript; p-phase, s-stream, T-trial	$\frac{kg}{h}$
$m_{p,s,T}$	Accumulated mass for specific subscript p-phase and s-stream, T-trial	kg
$\boldsymbol{m}_p$	Vector of reference masses for a specific phase-p $\in \mathbb{R}^T$	kg
$\boldsymbol{M}_p$	Matrix of accumulated masses for a specific phase-p $\in \mathbb{R}^{T \times S}$	kg
t_n	Time / timestamp / time segment point	sec
x_i	Mass fraction of a specific component *i* in a stream	$[-]$

1 Introduction

To date one of the most important topics of discussion in Norwegian oil and gas industry is digitalization. A large effort is currently put towards using digital technology, proper contextualization and unification of industrial data in to one common data repository. This will enable automation and more comprehensive data analysis, in order to enhance the business models established in the industry. Aker BP ASA [1] has joined in the establishment with Aker ASA [2] in the creation of the software company Cognite AS [3] to facilitate such a data repository, through the Cognite Data Fusion. Within the data storage repository, large amounts of historical sensor and aggregated values from sensors, monitoring and control systems is stored. This data is mainly used to look up previous states, but not prone to data science or any post calculation, This thesis develops on top of the data received from Cognite, which has simplified the development and access to the data streaming in from Aker BP's oil and gas producing asset.

Aker BP operates a floating oil and gas factory named Alvheim [4] in the North Sea. Alvheim initially produced from structures belonging to the field. Later in the operational development of the Alvheim asset, third party fields where tied back to Alvheim, utilizing the existing processing capacity and infrastructure. But the licensees and ownership fractions of the Alvheim field is not the same as on the third-party fields, where each field has a unique ownership split between the companies involved with development and operations. These fields are simultaneously producing oil and gas back to Alvheim, and the ability to accurately and correctly allocate the ownership of the produced oil and gas on Alvheim is of interest. This interest is not just for the involved companies but also with regards to calculating taxes to the Norwegian people. The third-party allocation measurements are performed on streams of oil and gas in the same pipe, in so-called multiphase streams with the use of multiphase flow meters (MPFMs), and these flow meters need to be calibrated regularly to ensure that the accuracy and representativity of the measurements are acceptable. And when it comes to the calibration method in use today, a significant deferral of production occurs during the calibration. This thesis will look into a more efficient method of calibrating these meters.

The main motivation is to be able to have no downtime for calibration runs of multiphase flow meters and better control of the health and performance of these meters, used in allocation of third-party fields. This is an emerging focus of both the Norwegian governments resource utilization of oil and gas deposits, as well as some of the oil and gas exploration and production (E&P) companies. To be able to effectively and elegantly allocate the ownership of oil and gas streams from multiple fields in order to utilize the existing infrastructure when the main oil fields are approaching their tail production[1] and capacity on the existing infrastructure opens. both in extracting resources from smaller oil and gas deposits normally not prioritized due to development cost together and providing mature oil and gas fields with additional oil and gas to process as well as production and transportation tariff opportunities [5].

The operations on the Alvheim ship have multiple times executed sequences of rerouting fluid streams over several days, to facilitate real data solely for testing the method investigated in

[1] Tale production refers to the decline curve in the later stages of the estimated production profile of an oilfield.

this thesis. The data is facilitated through Aker BPs infrastructure and contextualized and made available for this thesis through the Cognite Data Fusion (CDF) repository. Programing language Python [6] is used to create digital representations / digital twins of equipment in the fields as well as implementing this in an elegant codebase for execution of an algorithm. This solution made in this master thesis has the potential to solve an issue that is a highly relevant focus point of the Norwegian petroleum industry with regards to utilizing mature fields as mention earlier.

This thesis is organized in the following way:

Chapter 1: goes into the of the petroleum production industry in Norway, covering some historic moments, and about allocation and the hydrocarbon value chain, and expand some of the details with regards to governmental focus on third party fields already mentioned in the introduction.

Chapter 2: gives a brief introduction on fluids and essential concepts related to this thesis, regards to what goes on inside the closed containment of the fluid streams.

Chapter 3: gives a theoretical background of the sensors, instruments and measurements systems involved with creating the data used in allocation of oil and gas and is used within an algorithm created in this thesis.

Chapter 4: goes into creation of digital representations of streams, and how a traditional calibration is executed. Then the details of a parallel calibration method is developed, and ends with how this method is implemented into an algorithm that also utilizes the digital stream representations.

Chapter 5: gives results of calibrations executed on synthetic and real data

Chapter 6: discusses the results and the implementation of the algorithm, and the future development of the algorithm.

Finally, a conclusion will summarize what is achieved in this thesis.

Further technical details, complete results are covered in the attached appendices. There is one appendix which has the supplier documentation of the measurement system and data and results from offshore calibrations which will not be publicly available which provide a technical background and reports in the measurement systems used in this thesis, and not nessasery created by this thesis, but as a reference to non-public documents[2].

[2] Although the appendix is named appendix B, it will be found last in the appendices due to the nature of the document as a reference and not a product of this thesis.

2 Petroleum industry in Norway

This chapter goes into the history and some aspects of the governances, concepts concerning allocation of ownership of oil and gas and possibly essential aspects for the future of oil and gas in Norway.

2.1 History

In the 1950 few believed that there where oil and gas deposition on the Norwegian Continental Shelf (NCS), even The Geological Survey of Norway had even written this to the Norwegian Ministry of Foreign Affairs in 1958 that oil, gas or Sulphur deposits on the NSC was not very likely. But this would all change when a gas field discovery of the Groningen outside the coast of Netherlands in 1959, gains interest of the American oil companies, and gauges the question if there is more oil in the sea further north in the North Sea. And it was the American oil company Phillips Petroleum in 1962 whom sent an application to the Norwegian government to gain permission for exploration of oil and gas on the NCS. Which prompted the Norwegian government to develop the rules of governance for the potential resources on the NCS, underlining that the resources belong to the Norwegian people, and to be managed by the Norwegian government. [7] The 13th Of April 1965 the Norwegian government under the Ministry of Petroleum and Energy (MPE / OED) gave concession to Petroleum Exploration and Production (E&P) companies to explore the NCS for petroleum deposits. [8]

2.2 Licenses and Blocks

The concessions / licenses given to companies where limited to geographical areas called blocks. The administration of these concession blocks where first done by the MPE but in 1972 the Norwegian Petroleum Directorate (NPD / OD) was established to function as a specialist directorate and administrative body of the oil and gas activities, together with the creation of the governmental oil company Statoil, now known as Equinor. Later in 2004 the NPD where split in to two, where the safety and work environment of Norway's petroleum activities where to be administrated by Petroleum Safety Authority Norway (PSA / Ptil) and the NPD would continue with the resource management of the petroleum activities in Norway as well as serve as advisers to the MPE. [9].

Back to these licenses, they are given during licensing rounds and give the E&P companies the opportunity to explore the block area for resources. Each geographical block contain multiple licenses where there are several E&P companies splitting the risk, ownership and development cost of exploration and production between the licensees of the block. And if found the potential development of a field can start. And within development phases the choice for infrastructure and measurement and allocation solutions is initially decided, which is a topic relevant for this thesis.

2.3 Ownership, production metering, allocation and hydrocarbon management

In simple terms an upstream petroleum producing asset such as Alvheim, has Oil and gas streaming up from a well in a petroleum reservoir. The well fluid is processed by removing unwanted fluids such as water and prepared for transport through pipelines or shuttle tankers to further refining and the market. Figure 1 shows an example of a value chain for a E&P company[3].

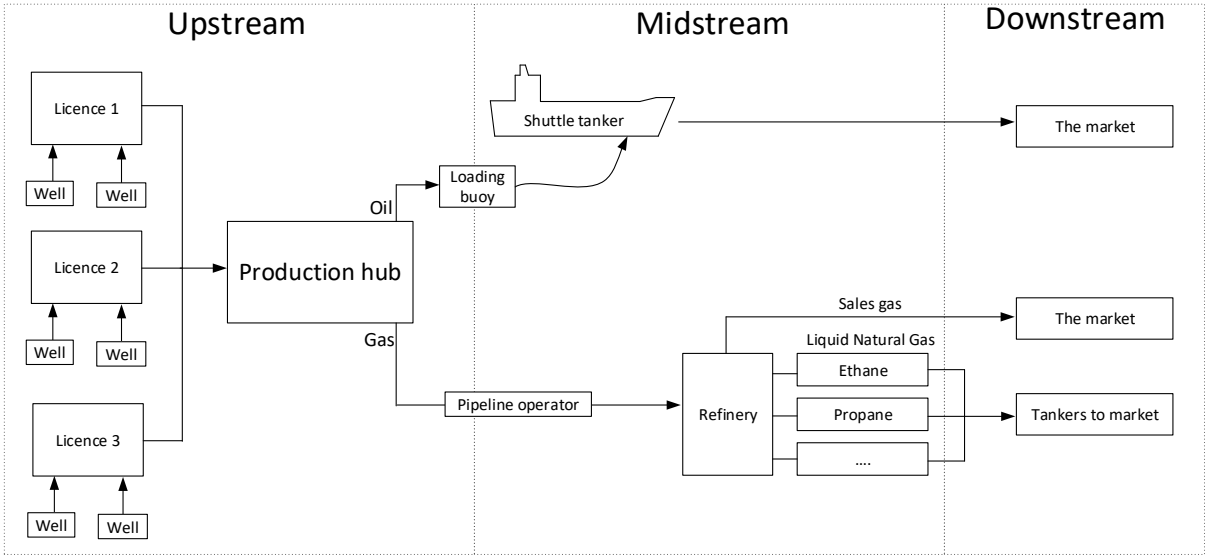


Figure 1 Hydrocarbon Value Chain

2.3.1 Allocation

Consider an oil field constantly producing oil and gas, the quantity of petroleum produced has to be continuously measured and counted, and this is where the fiscal metering and allocation systems comes into play. Metering systems constantly measure and count the production, and aggregated data into daily and monthly production. And theses daily amounts are then further allocated into the different licenses where the hydrocarbons are extracted.

2.3.2 Measurement and measurands for allocation of oil and gas.

Looking at the streams of hydrocarbons produced, in petroleum engineering literature [10] the stream can be quantized in many different units such as mol compositions and rates, volumetric units such as m^3 or barrels of oil, and standardized volumetric units, such as Sm^3 / stock tank oil, or even energy in units such as barrel of oil equivalent (BOE) or pure energy content in Joule. But in metering and allocation and this thesis the mass flow rates and composition given in mass fractions will be used. Mass is a general standard to allocate production into transport systems, as well as it is a common unit for interdisciplinary

[3] Oil pipelines with non-stabilized oil where the condensates are separated at a refinery / terminal are also a common midstream infrastructure for hydrocarbon transport.

engineering fields, and simplifies calculation by removing concepts such as pressure- and temperature-effects as well as compressibility of the different fluids and fluid phases. It is simpler and more elegant to work with mass rates and mass fractional compositions to explain the concepts. But allocation can be performed on any extensive property concerning the physical quantity such as volume, energy, mass or substance amounts in moles. It all depends on the measurement executed to achieve the calculated amount as well as the extent of the instrumentation of the allocation measurement. Uncertainties are specified for both the entire measurement system as well as uncertainties for individual measurements as specified by the NPD in Section 8 of *Regulations relating to measurement of petroleum for fiscal purposes and for calculation of co2-tax (the measurement regulations).* [11] If the measurement system is to deviate from practices in the NPD's regulations this needs to be clarified by the NPD. Figure 2 depicts the concepts of a production asset covered in this chapter, as well as some mathematical notations and calculations which will be covered later in this thesis and its appendices.

2.3.3 Field Blend

The characteristic blend of hydrocarbons from the reservoirs from a specific petroleum producing unit. Each well in each field has its own characteristic composition and by combining each of the fluid streams in a production hub, a generic field blend is created. Each individual component in the blend have a specific market value constantly changing due to the dynamics in the economics of the petroleum market.

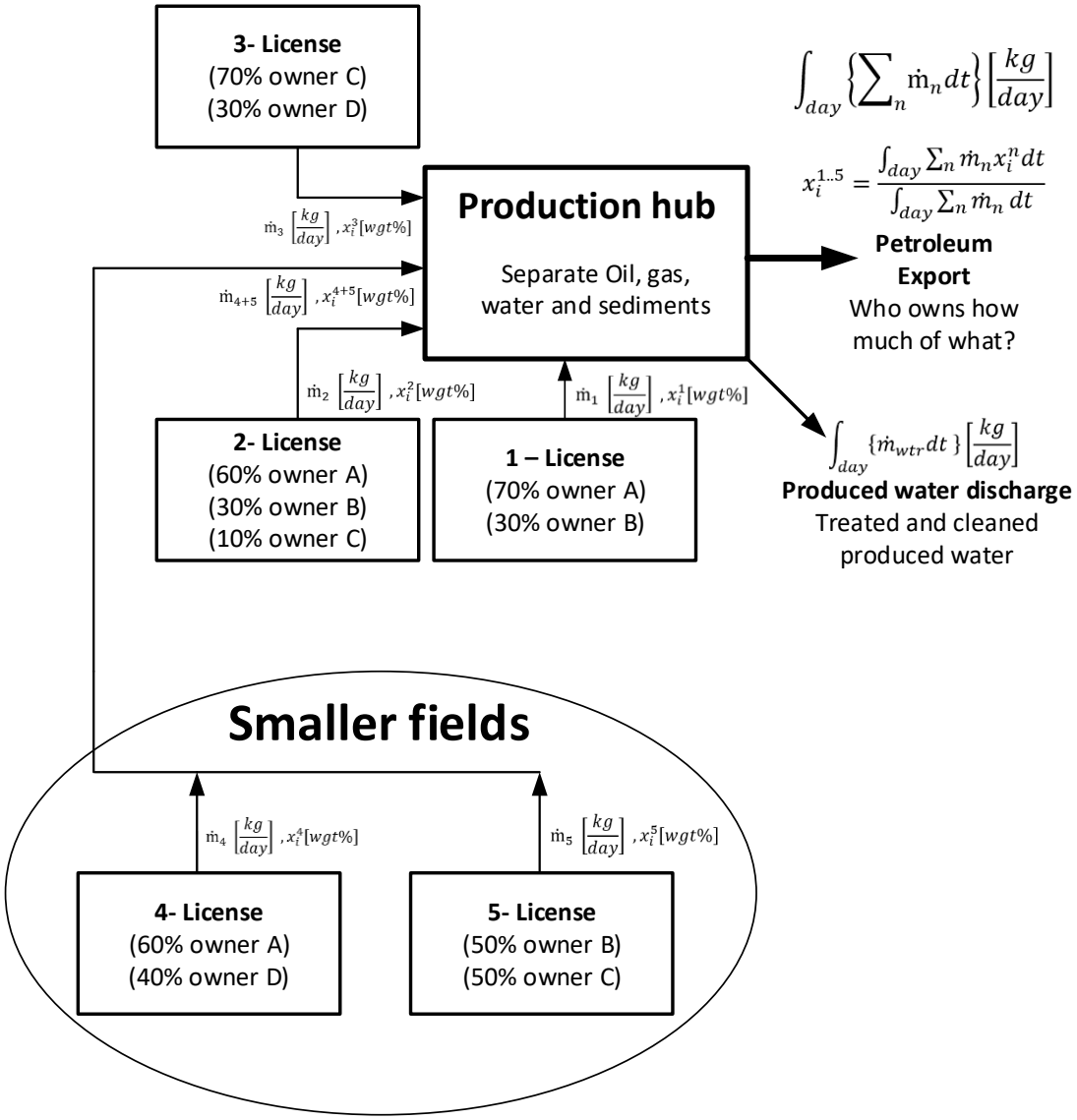


Figure 2 - Petroleum Field Allocation by mass example associated with different license agreements

2.4 Future prospects and focus on the Norwegian Continental Shelf (NCS)

In order to have a sustainable oil and gas industry, in the developed part of the North- and the Norwegian-sea the focus of some the E&P companies is to utilize the existing infrastructure to develop and tie-in of oil fields to existing production hub, so called third-party fields. in order to extract oil and gas from these. And all these smaller licenses / fields usually have unique ownerships allocated to the fields. So, production allocation from each of the smaller fields has to be measured somehow. [5] this focus is so important for resource extraction the MPE have made a regulation relating to the use of facilities by others called the Third-party Access (TPA) Regulation in 2006 [12]

3 Separation and flow of fluids - Brief theoretical background

This chapter goes into the theoretical background to familiarize the reader with the main concepts that are essential to this thesis. Concepts such as petrochemistry, phases, compositions and the balance laws which govern the dynamics of the system in question.

3.1 Hydrocarbon fluid and separation

The fluid flow from a well consist mainly of hydrocarbons, nitrogen, carbon dioxide and base sediments and water and hydrogen sulfide [10]. In the gas phase, hydrocarbons dominate the composition. But the liquid phase is primarily separated by oil and water, hence polar and non-polar liquid.

The hydrocarbons are non-polar and the water is polar, this due to the water molecule is a dipole, the water has also hydrogen bonds with other water molecules and thereby giving it a higher density than liquid hydrocarbons and a higher boiling point than the majority molecules of similar nucleic weight such as the lighter alkanes / paraffins of the hydrocarbons, but this boiling point and density difference is important later when it comes to the processing and refining of a hydrocarbon fluid stream.

Consider a closed container with a mix of a well sample fluid; the gas floats to the top, the oil separates in the middle and the water collects at the bottom of the container vessel. In reality on the other hand, the gravimetric phase settling is done in big vessels called separators. These separator vessels have a continuously changing stream of fluid in and out of the separator. Within the fluid mix in the separator, there is a layer of emulsion of both oil and water, and this emulsion require time to separate into either oil or water, And the speed at which this emulsion separation can occur is dependent on temperature and chemicals aids that helps breaking the emulsion in to two separate phases.

3.2 Fluid flow

Different streams can be measured and allocated in different units as covered in 2.3.2. Expanding upon fluid flow; flow as stated earlier can be quantified in many units, but most commonly fluid flow is in a volumetric unit, which is the mean velocity of a fluid multiplied by the cross-sectional area of the conduit the fluid is flowing in, as depicted in Figure 3.

When it comes to the application in this thesis the conduits in question can have multiple phases at the same time. Because fluid streams of hydrocarbon in a pipe conduit, vessel / containment of the hydrocarbons in upstream oil and gas producing facilities are mainly in two aggregate phases, which is in liquid and gas. and in the case of this thesis there are both conduits of a single phase and also conduits where there are multiple of phases at the same time, where there are both liquid and gas flowing through the same conduit at the same time. and the fraction of each phase are in constant motion. But for hydrocarbon streams a rule of thumb is that; the closer the fluid stream is to the well in the reservoir the higher is the chance, of it being a multiphase stream.

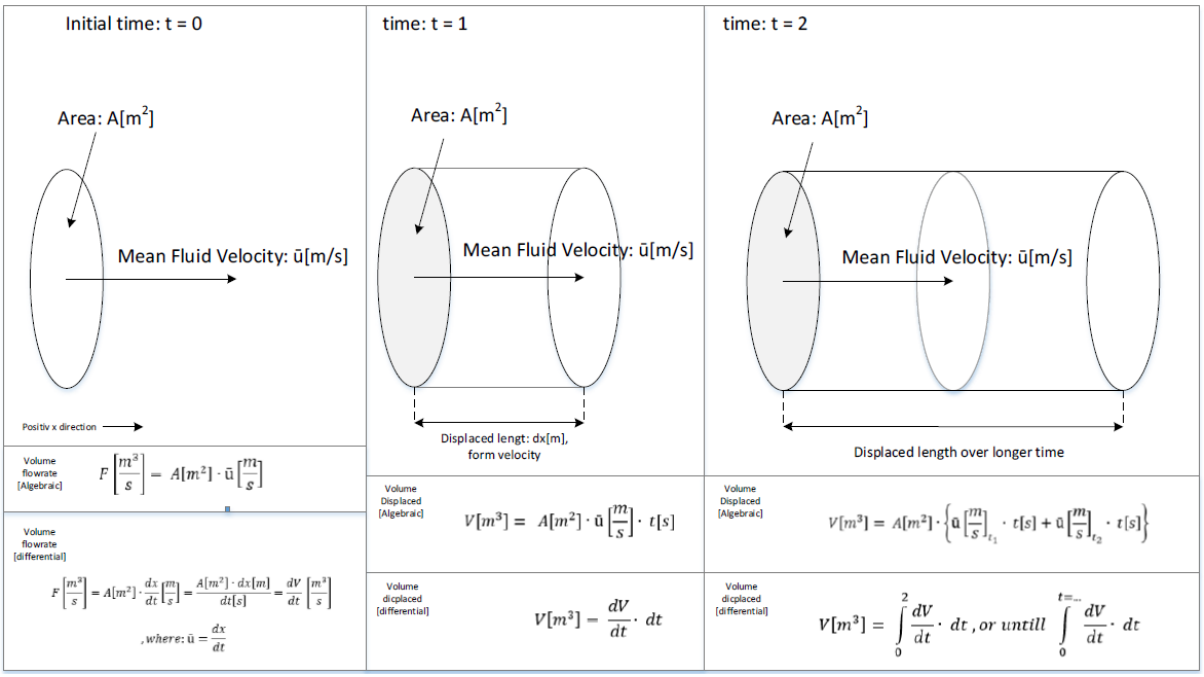


Figure 3 - Concept of volumetric flow and discrete accumulation, where is u velocity and $\bar{u}$ average velocity [13]

3.2.1 Phase separation

The initial separation of the phases of the multiphase well stream is most offend done in an inlet separator shown in Figure 7. This is done primarily done through gravimetric separation of the different phases, where the mass density of the different phases separates the different fluid fraction where the heaviest phase is collected at the bottom and the least heavy at the top.

3.2.2 Single-Phase flow stream

The separators effluent streams are all mainly single-phase flow streams, and in this thesis means that the gas outlet has only gas, the oil outlet only have oil flow and the water outlet stream only have water. In reality gas bobbles can occur in liquid phases, and liquid droplets in the gas phase, and residual water in the oil stream and residual oil in the water stream, due to not enough settling / retention time inside the separator vessel.

3.2.3 Multiphase flows stream

Physically there are mainly to aggregate phase differences which is liquid and gas , when it comes to multiphase flows such as the multiphase streams in the topside Third party MPFMs on Alvheim, it is important to split the liquid in to two phases as well, since there is a significant amount of water in the stream. Therefor the liquid phase is differentiated into oil and water. When characterizing a multiphase flow one of the main variables within multiphase flow is the gas volume fraction (GVF) which is the gas volume flowrate relative to multiphase volume flowrates at the pressure and temperature conditions in the conduit

section in question [14]. And when it comes to multiphase flow meters their uncertainties are categorized according to the GVF the meter is measuring.

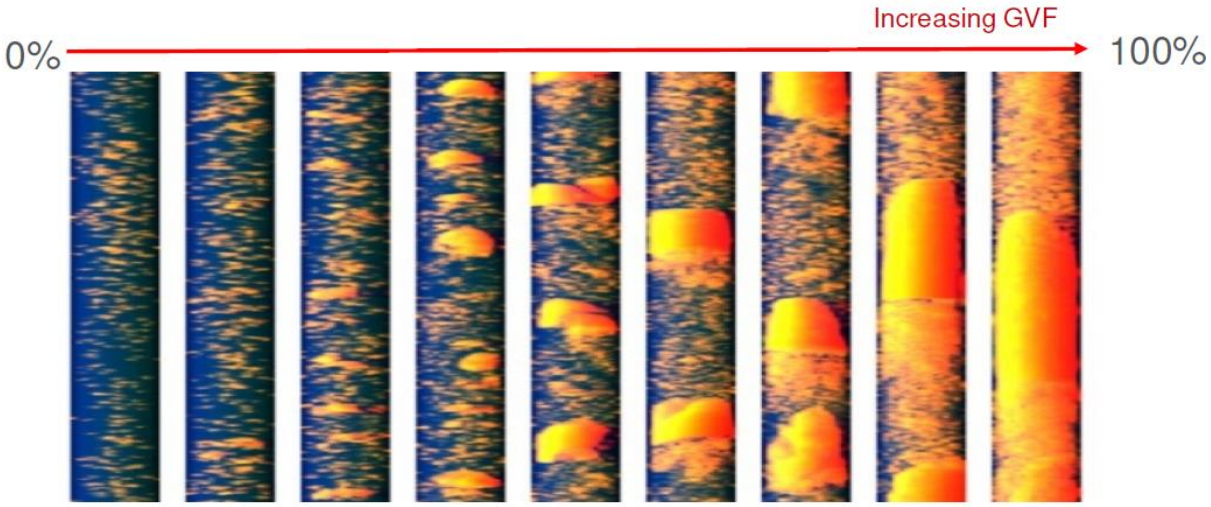


Figure 4 – Multiphase flow with increasing GVF, where the blue part is liquid and the yellow represents gas and gas bubbles [15]

In addition, when looking at a two-phase stream, with liquid and gas in a stream there are more than the GVF to look into, such as variables for understanding the flow regime of the stream. These are the superficial velocities, which are the volumetric flowrate of the specific phase, divided by the cross sectional area of the pipe / conduit, Figure 5 from the MPFM handbook [14] shows the different flow regime occurring in a vertical pipe such as the multiphase meters are on Alvheim. Each flow regime gives rise to its own challenges when it comes to precisely measure the multiphase stream during different conditions, and each regime has a to a degree a unique uncertainty dependent on the velocity of the media.

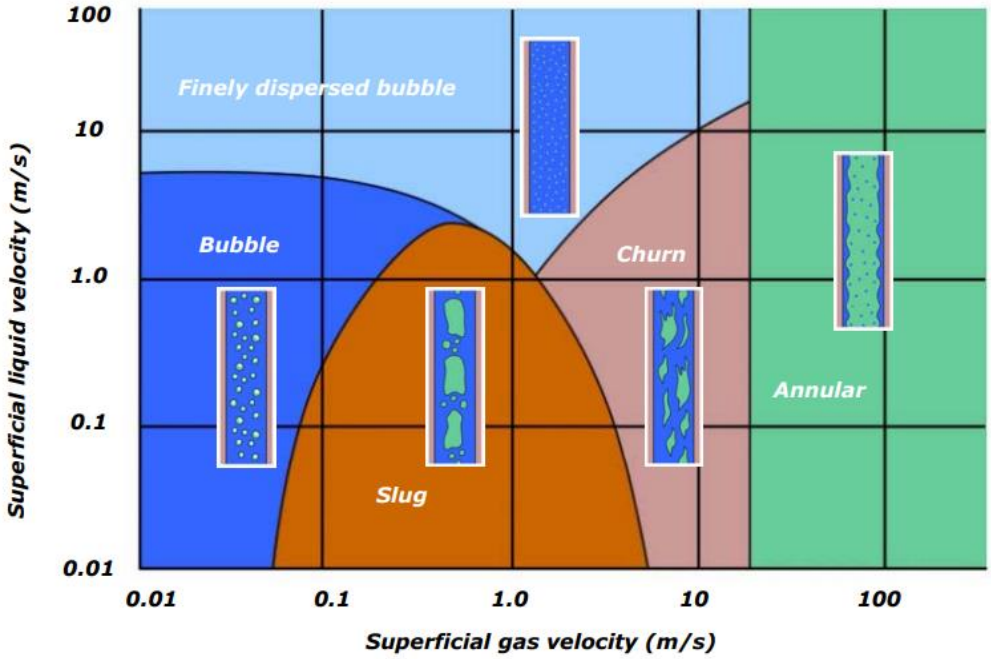


Figure 5 - Two-phase flow map of a vertical pipe [14]

In the intermittent flow regime there can occur elongated baubles causing occasionally a multiphase stream to have liquid and gas plugs in the axial direction of the pipe, and over a time period the multiphase stream can go from being fully displaced by liquid to being fully displaced by gas, this phenomena is called slugging, and cause challenges for measuring and controlling processes involved with multiphase flows[4].

3.2.4 Accumulation of flow

The task of estimating how much has flown through a conduit from measured flowrates from a flow meter is very important. The measurement needs to be accumulated to go from a rate to a quantity. This accumulation is done by integrating the flowrate measured by a flow meter with respect to time. Equation (3.1) shows this for a volumetric flow to a volume and a mass flow to accumulated mass in two separate differential notation where the first term is the Leibnitz notation and the second is Newtons notation. And is displayed in the volumetric case as the displaced volumes in Figure 3.

$$\int_{t_0}^{t_1} \frac{dV}{dt}\,dt = \int_{t_0}^{t_1} \dot{V}\,dt = V\,, \qquad giving\ the\ units\colon \left[\frac{m^3}{sec}\cdot\ sec\right] = [m^3]$$
$$\int_{t_0}^{t_1} \frac{dm}{dt}\,dt = \int_{t_0}^{t_1} \dot{m}\,dt = m\,, \qquad giving\ the\ units\colon \left[\frac{kg}{sec}\cdot\ sec\right] = [kg] \tag{3.1}$$

[4] Slugs also cause water hammers that stress and cause vibrations on the system, this due to the sudden change in the inertia of the mass flow, especially in bends where the inertia of the mass flow exert force towards the pipe bend wall in order to change the mass flow direction

3.3 Balance laws

The main balance law governing the main physical properties in the calibration of the MPFMs is the mass balance of fluid in the system. For a given period of time the mass flow in to the system must be equal to the mass flow out of the system, if they are not equal this states that there is some collection or accumulation of mass some place in the system, which can be gauged through changes in state changes in levels in a tank, or mass-density changes due to temperature and pressure changes within the system. Over time the accumulated mass flow in to the system must equate to the accumulated mass flow out of the system.

3.4 Modeling of the dynamic phenomena

The systems in question are subjected to flashing of hydrocarbon components from a liquid state to a gaseous state in the separator, mostly driven by a reduction in pressure or an increase in temperature

3.4.1 Separator modeling; balance laws and phase mass exchange dynamics

Modeling the states inside a phase separator vessel can be very complex, but for the sake of the purpose of this thesis the mass balance will be of focus in this thesis. But if only the liquid level was to be modelled the liquid volume balance could do this to an often-satisfying degree. The mass balance law states the sum of all mass flow influent to the system must be equal to the amount of mass leaving the system, if they are not the same there is a change of the total mass in the system, as stated in equation 3.2.

$$\frac{dm}{dt} = \sum \dot{m}_{influent} - \sum \dot{m}_{effluent} \tag{3.2}$$

By accumulating the mass flowrates in (influent) and out (effluent) of the system, over a long enough time, the sum of both of these accumulated values should approach zero. But due to dynamics and uncertainties in measurements of the rates influent and effluent of the system, in addition to the total mass stored in the system can be potentially changing. But in order to ensure that the mass rate in to the system equals the mass rate out of the system. The stored mass inside the system needs to be gauged somehow. The separator has sensors measuring the liquid oil level, and the liquid interface between oil and water, as well as the pressure. And if the levels and pressure inside the separator are stable during the time window in question, an assumption that there is no loss or accumulation of mass inside. Thus, the statement that the mass rate into the system, equals the mass rate leaving the system, and hence give grounds for a comparison between a multiphase stream toward single phase streams. there are constant feedback controller controlling the oil water interface, oil level as

well as the static pressure in the vessel. This mass rate comparison is also depicted block schematically in Figure 6.

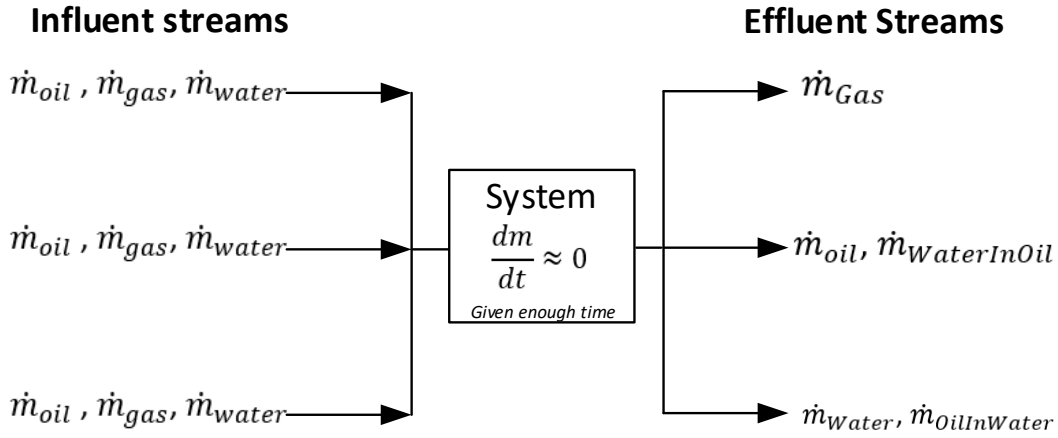


Figure 6 - Block schematic abstraction of mass balance

Figure 7 depicts the mass balance and dynamics occurring in the calibration system. Where there are three multiphase streams upstream the separator with oil, gas and water mass rates from each multiphase stream, and there are single phase streams effluent of the separator. As well as a simplified mass-based flashing dynamic, which occur due to change in pressure and temperature from the multiphase meter and the separator. In this thesis the applied flashing dynamic is simplified to a mass fraction of the oil flashes to gas mass phase. Which is based on values from a process simulator, this result is shown in Appendix B.

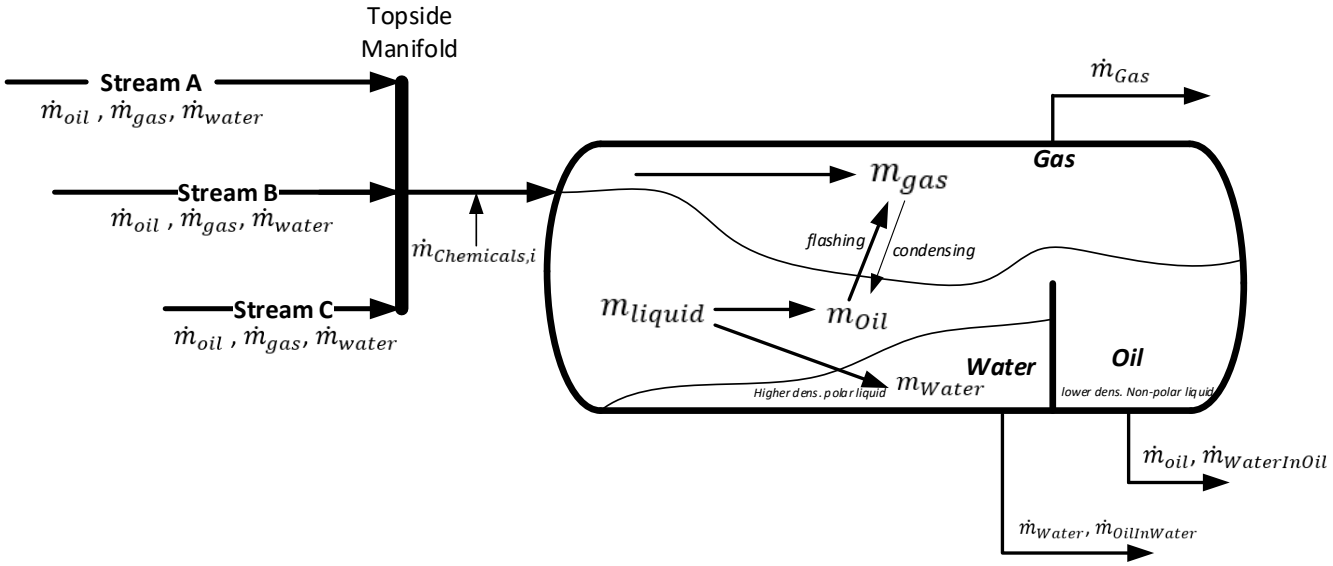


Figure 7 - Mass balance of multiphase streams and separator

3.5 Flashing, PVT and Phase equilibrium

Every stream has a unique composition changing over time but most essentially for the hydrocarbon components in the stream, when there is a reduction in pressure or an increase in temperature a fraction of the lighter components turns from a liquid state to a gaseous state. Inside the separator, the fluid also has a chance to settle for a short time. The flashing of liquid oil is due to a higher pressure and temperature than the bobble point of the specific components, turning a fraction of the liquid component into a gaseous state. The liquid will continue to flash until the containment of the liquid has equalized the pressure in the containment to the specific vapor pressure for the given pressure and temperature. There is also an effect which goes the other direction (from gas to liquid), which is condensation of gas to liquid droplets, where a fraction of the gas is approaching the components dewpoint. These phase changes are approaching the phase equilibrium of the fluid, where the phase fractions of the fluid change with pressure and temperature for a given composition. Hence every valve, heat-exchanger, pipe bend, and length of pipe causing a pressure or a temperature to change and changes the phase equilibrium point of the fluid and thus the phase fractions. This is important due to the multiphase meter and a single phase stream are separated by pipe and other processing equipment and have different pressures and temperature and thusly different phase fractions.

4 Technical background of hydrocarbon flow metering

In the real physical world, a requirement to be able to quantify the extensive and intensive variables of both a multiphase stream and separated single phase streams. This chapter will dive conceptually into instrumentation and calculations used to gauge these variables, which are subsequentially used to perform ownership allocation and calibration. This chapter will try to provide information on the technical aspects of both the physical and computer systems used in the creation of the data used in this thesis. The chapter will also go in to aspects of the calibration and traceability of the measurements to international standards. In essence this chapter's purpose is to get an understanding and appreciation of how continuous measurements from instruments are combined into a elegant and purposeful symphony.

4.1 Instrumentation

From the definition in section 2 of the measurement regulations from the NPD an Instrument is defined as:

"*An assembly consisting of a transducer and one or more sensing elements. The signal from an instrument represents a physical condition. A technical device used to measure a physical parameter*." [11]

In essence a instrument is a piece of equipment which measure one or more physical parameters. And when this measured physical parameter is used to a fiscal purpose which is that there is some form of ownership or monetary transactions based on the measurement then the measurement is consider a fiscal measurement. When the measurement effects the owners, operators and government the involved parties will naturally require a quality assurance, that the measured parameter is relatable to known physical quantities with a given uncertainty, this operation to compare a measurement to a known reference is called a calibration, where the characteristic uncertainty of the measurement over the measurement range of the instrument is established. Sometimes confusion arises when it comes to the difference between calibration and adjustment, but calibration is establishing the uncertainty and adjustment is when a physical action is made to change the output of the instrument.

Instruments such as pressure and temperature will not be covered, but their importance in the measurement is not to be neglected. Pressure and temperature have a important function when it comes to standardizing / normalizing the volume rates to common conditions, and in the section 10 of the measurement regulations from the NPD this standard conditions are set to be 101,325 kPa and 15 °C [11], and this calculation is performed at each fiscal flow and density measurement in order to gain standardized and thus comparable figures.

4.2 Liquid flow measurement

Most typical liquid flow meters in general try to their best to measure the mean fluid velocity inside the meter body. And by having a known cross-sectional area in the meter body, the volumetric flow becomes the product of the area and the mean fluid velocity, this applies to almost all of the flow meters used except for the Coriolis meter witch directly calculates the

mass flow rate. But inside the systems in question inside this thesis the Coriolis meter is not used, but it is in regular use as both a process control and measurement applications. The main single liquid meters used in this thesis is ultrasonic flow meters (USFM), and Turbine meters for the oil stream, and electromagnetic flow meters for the water stream.

4.2.1 Oil flow measurement

Some flow meters are more suitable for measuring oil flows more than others, but in general the most used single-phase oil flow meters for continues flow measurements are Turbine-, ultrasonic- or Coriolis-meters. And due to that hydrocarbon oil is nonpolar some meters are not used for this liquid.

4.2.1.1 Turbine flow meters

Turbine flow meters uses the kinematics of the fluid particles to rotate a turbine inside the pipe, and on fixed locations on the meter body there are place magnetic pickups which produce a pulse when a turbine blade rotates past the pickup. Over time this generates a pulse train, where the frequency of the pulse train is proportional to the volumetric flow rate. And this proportionality is mostly implemented as a fixed calibration factor called the K-factor or sometimes the meter factor, with the units $\left[\frac{pls}{m^3}\right]$. The k-factor is not necessarily fixed for the entire flow range the meter is operating on and sometimes there is a calibration curve where the k-factor is a function of the frequency of the pulse train or the measured flowrate.

Diagnostics

On the turbine meters used for custody transfer it is normal to have two pickups place with a fixed angle between each other in order to generate two pulse trains with a fixed phase between the pulse trains. And if a turbine blade is damaged or some other issue occur there is a diagnostic alarm raised if the phase between the pulse trains is not as it should be, and an indication of the direction of flow is also inherent in what pulse train is leading.

4.2.1.2 Ultrasonic flow meters

Ultrasonic liquid flow meters use sets of ultrasonic transducers inside the meter body to probe the velocity of the media, by measuring the time of flight of the ultrasonic pulse propagating through the media in the meter, where the fluid flow inside the pipe increases the time of flight if the pulse is traveling downstream or decrease the time of flight if the pulse is traveling upstream. And by having multiple sets of these ultrasonic transducers placed in the meter body at strategic location a flow velocity profile (flow profile) can be established. And by calculating the mean fluid velocity from the flow profile and multiply the velocity with the know cross-sectional meter body area, the volumetric flow measurement is established.

Diagnostics

The most significant diagnostic or beneficial measurement is that from the time of flight data the speed of sound in the media can be calculated. This is especially helpful for gas flow applications through ultrasonic flow meters where this speed of sound can be compared with the sound speed calculated from the measured or expected composition and conditions through the AGA-10 report. The measurement can also potentially be used to infer density from an expected composition and the speed of sound measurement provide additional

correcting data. Additionally, the diagnostics from the physical and the signal processing of the raw transducer data such as the gain of the transducers to probe the health of the transducers, and the signal to noise ratio, giving a lot of information about the internal health of the meter.

4.2.1.3 Produced water flow measurement

Measuring the produced water outflow from the separator gives rise to utilize the ions from the salts and impurities which gives water its electrical conductance. Electromagnetic flow meters are frequently used for measuring produced water. The electromagnetic flow meters work through electromagnetic induction, by having magnetic coils around the meter body, to create an electromagnetic field around the pipe, electrodes in contact with the fluid on opposite sides of the meter body. When the media in the pipe is conductive and moving through a magnetic field a voltage is induced in the liquid, through faradays law, which is then measured by the voltage difference over the electrodes in the media.

4.2.2 Calibration and traceability of liquid volume flow meters

The liquid flow meters discussed in this chapter are all volumetric flow meters, and when calibrating and establishing the uncertainty and linearity / characteristic of the flow meter, this is historically done in a prover-loop with the ability to displace a fixed volume through the meter and comparing the displaced volume with the measured volume. the liquid volume flow rate is for oil-metering application is electrically sent as a pulse train, where the frequency of the pulse train is proportional to the volume flowrate. And when calibrating a meter the calibration factor can be established through counting the pulses on the pulse train during the displacement of the fixed or known volume[5] the example of using a compact prover unit as the displacement reference volume shown in Figure 8. By repeating the fixed volume displacement through the meter, the calibration factors is then established if the measurements point achieve a sufficient confidence degree calculated through the student-t probability density distribution, specified in API-MPMS-4 Proving System. This exercise of establishing calibration factors can then be executed at varying conditions, mostly significant with varying flowrates, and from this the linearity or non-linearity of the meter is established and if a single calibration factor is sufficient for the liquid flow meter or if it requires a calibration characteristic, where the calibration factor dependent on flowrate or any other measured varying condition.

The prover loop for most applications is placed inside a flow laboratory, but on fiscal metering stations are often fitted with an in-situ prover loop, where the NPD requires the calibration / proving to be done every 4th day or if the fluid properties change from last prove. [11]. But the meters used on the liquid streams on the Alvheim third party separator are all meters and calibrated in a non-bias 3rd party flow laboratory at regular intervals.

[5] This fixed or know volume is also known as volume-normal

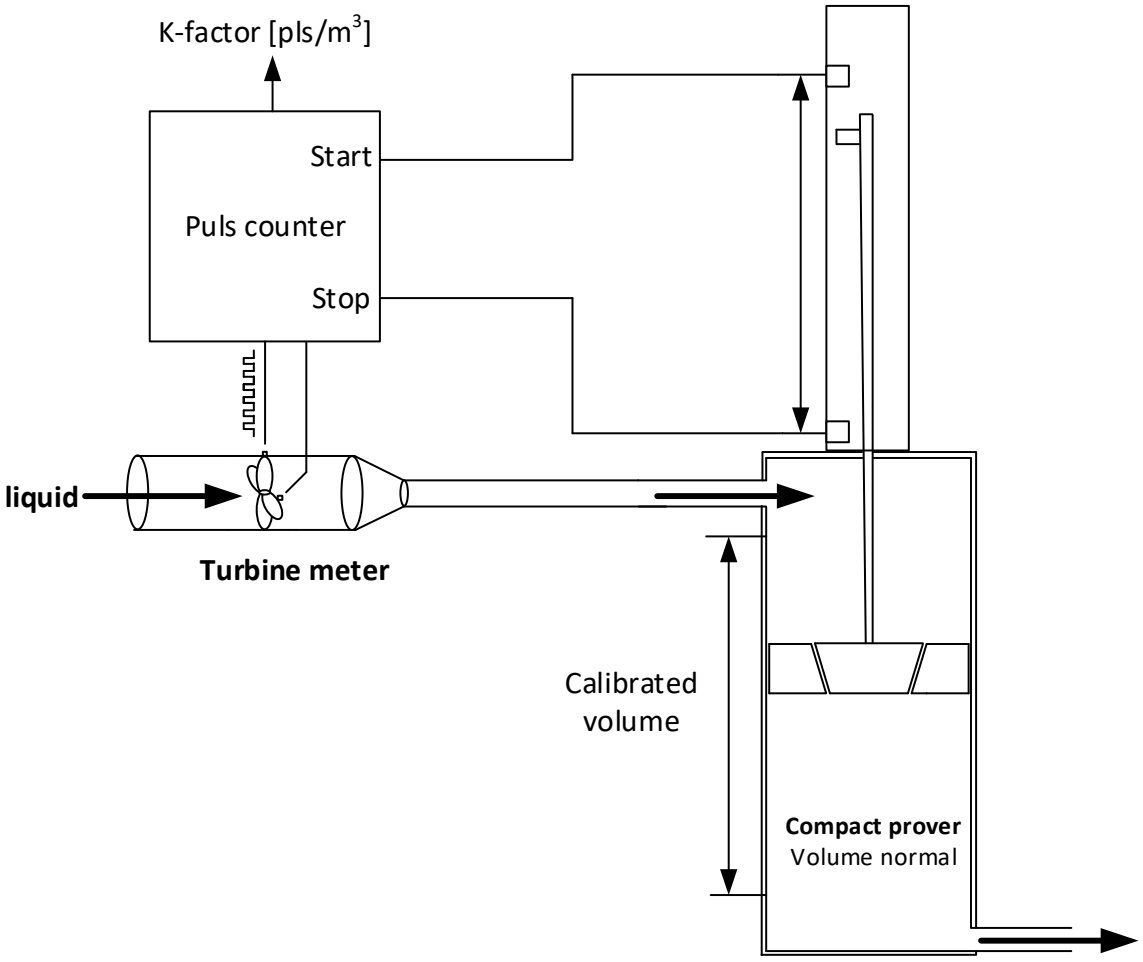


Figure 8 – Liquid turbine meter flow calibration with compact prover

Establishing the calibration of the fixed displaceable volume (prove volume) also needs to be able to be checked against a higher order reference, this is done by displacing the volume normally with water into a calibrated can (seraphin can), which is purposely built to gauge the volume inside the prover. This seraphin can is then calibrated by mass, by filling the can with water with a known density. And measuring the mass of water inside the seraphin can, the measurement has then changed from a volumetric measurement to a mass measurement, and by weighting the mass with a measurement device traceable to the kilogram standard in France, a traceable standard for the calibration of volumetric liquid flow meters is complete. This traceability chain is depicted in Figure 14.

4.2.2.1 Auxiliary measurements of intensive variables

Densitometers

To get a volumetric flowrate to a mass flowrate the mass density has to be measured, and this is done through a density transducer or densitometer which in most traditional cases for precise measurement is done through a oscillating tube, which is agitated by electromagnetic coils, giving the measurement tube a vibration. The frequency at which this tube is oscillating correlates to the mass within the tube system, and the heavier the mass within the tube is the

lower the frequency at which the tube is oscillating, and thus a density reference is measured through a characteristic of time period of oscillations.

Water cut meter

Water cut meter is another word for instrument which measures amount of water in oil in water / the water in oil fraction if you prefer, and the standard measurement principle used here is a microwave based measurement principle, where the characteristic damping of a microwave signal propagating through a oil water mix at different microwave frequencies, a relative characteristic attenuation over a frequency span, is then effected by the water content in the liquid mix[6]. And for increasing the accuracy the density from a densitometer is also used as an input to this measurement. More details of water fraction metering can be found in a handbook on the topic from the NFOGM [16]

4.3 Gas flow measurements

When it comes to gas flow measurement in recent years, the standard flow meter primarily used in new applications today is the ultrasonic flow meter. The ultrasonic flow meter in liquid flow measurement chapter is analogous to the one used in gas flow measurement. There is one essential difference between liquid and gas applications, which is that the piezo electric transducers are installed with direct contact to the gas and not in a separate pocket/well which is isolated from the media.

Other measurement principles where also used such as differential pressure gas flow measurement; gas flow measurement through differential pressure over an orifice plate or any other reduction of cross-sectional area of the conduit. This reduction in area intern cause a change in differential pressure due to an alteration of the kinetics inside the conduit. All this is based on the age-old Bernoulli energy balance equation in iso-metric form, which is further enhanced and the development into ISO-5167 series for gas flow measurements with a higher precision and higher statistical confidence.

Gas flow meters are calibrated in a similar manner as the liquid meter calibration, a gas flow meter can measure the displaced gas, but instead of a volumetric displacement the displacement has to be converted to a mass / standard volume displacement. In order to calibrate the gas flow meter since gas is compressible, a mass flow comparison has to be performed, and instead of having a volumetric displacement standard volume or a mass displacement reference is needed for the gas flow calibration, but the methodology is the same[7].

[6] in the same manner that a microwave oven heats up / agitates the water molecules / moisture within the oven.

[7] Calibration of gas flow meters are not frequently done, mainly prior to install or on specific occasions, meters are sent to special laboratory's such as FORCE's calibration facility in Denmark or EuroLoop in The Netherlands, but there is a limited number of facilities providing this service, liquid flow laboratories are more frequent.

4.4 The computer part[8] – Flow computer

On each of the measurement streams its own embedded flow computer is installed. The flow computers task is to collect and distribute measurements from instrument, execute sensor calculation and accumulating and storing the increments calculated by the flow computer. these accumulators in P&ID notation are usually denoted Flow Quantity Indicator (FQI), and also in other process context referred to as totalizers. The calculation for each increment is done every second, for the single-phase streams and every 10 seconds for the multiphase stream computers before the increment is stored inside the flow computers database. Figure 9 shows a representation of the liquid oil stream, with instruments and a fast loop, and the abstract calculation performed by the stream flow computer. The flow computer also works as a controller for the supervisory metering system, and can usually control the connected line valves through the flow computers I/O. The system specific details of the flow meters and their exact internal calculation are in appendix B.

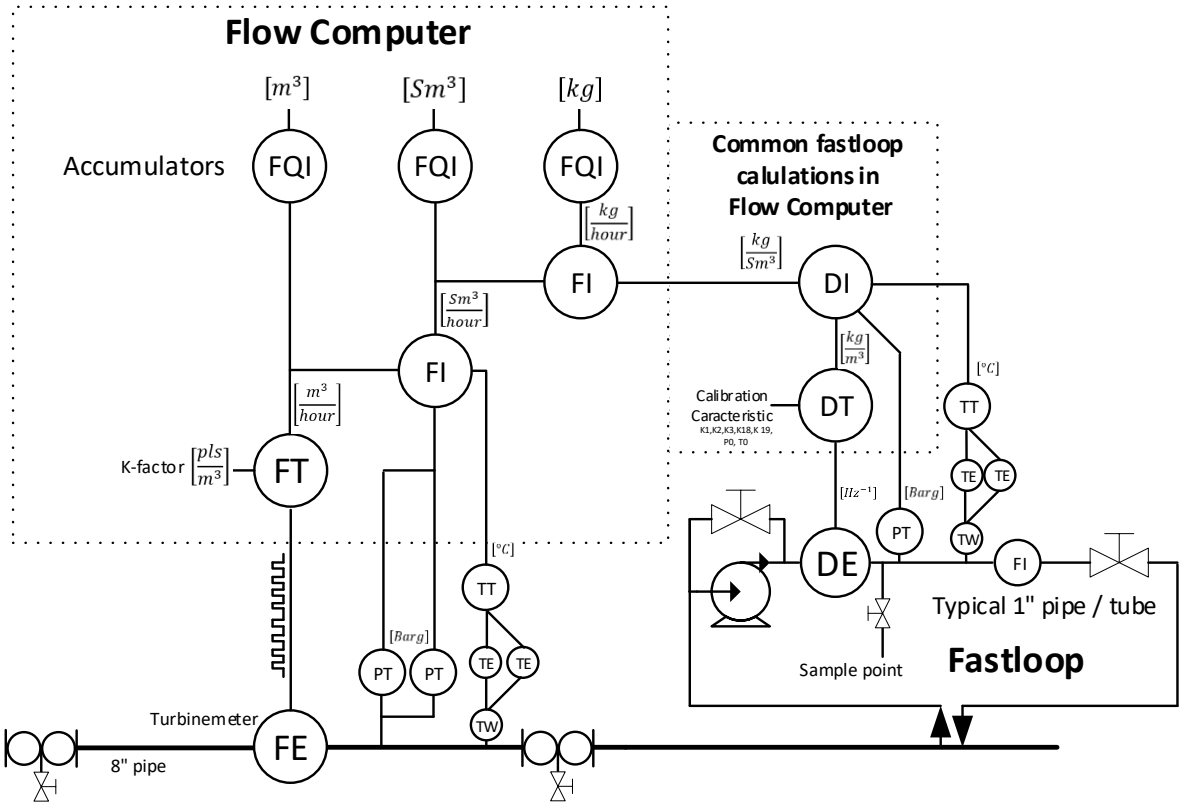


Figure 9 - Overview of typical and simplified single-phase fluid measurement stream with the abstract tasks and calculations of the flow computer

[8] The computer part refers to the definitions in the NPD's measurement regulations [11], but is the control and computational processing part of the measurement system.

4.5 Multiphase flow meters

When it comes to streams where there are multiple phases and the meter used is an inline multiphase meter, which has instruments working closely together to establish the flow of each of the phases, and these inline meters are used on the Alvheim installation.

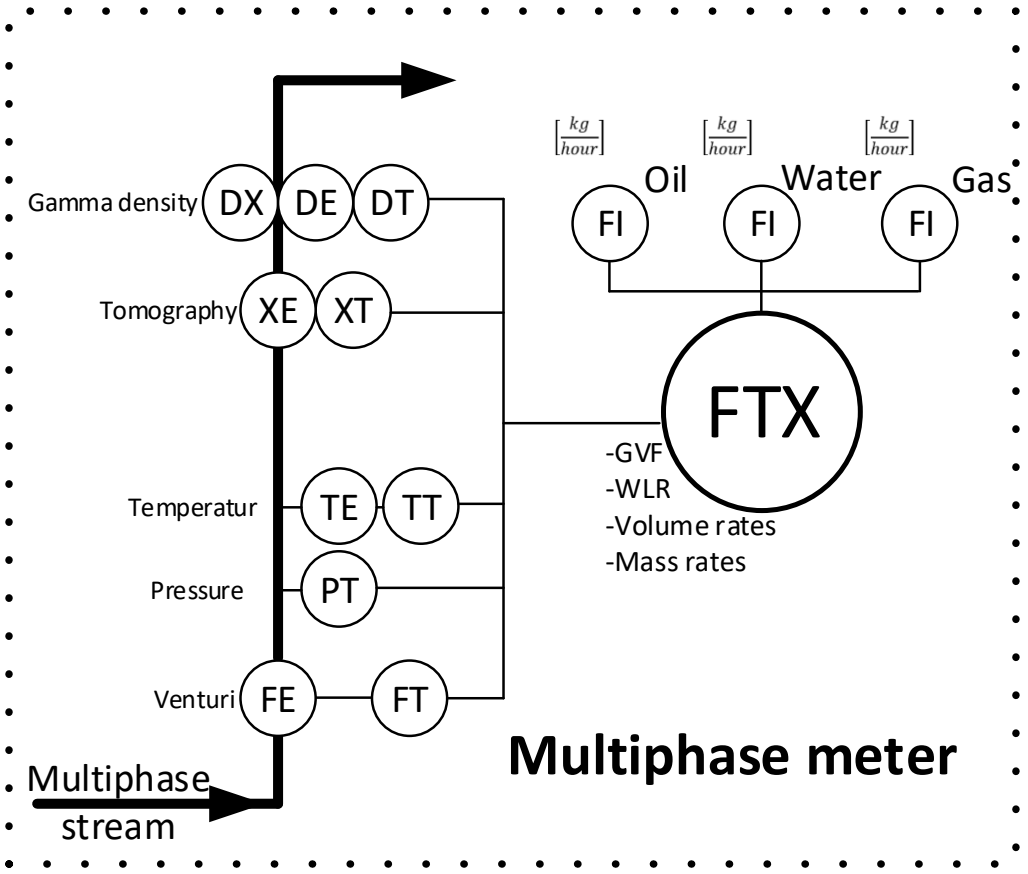


Figure 10 - Conceptual overview of instruments involved in a multiphase meter

On the Alvheim platform the multiphase flow meters from MPM, now owned by TechnipFMC will be the focus here. But there are many other multiphase meters and solutions on the market, more information on multiphase flow meter selection, solutions and uncertainty see the handbook from NFOGM on the topic [14] But for this thesis the focus when explaining the multiphase meters will be on the MPM meter from Technip FMC.

But multiphase meters are a lot more complex than a normal single-phase meter. This due to within fraction of a second the fluid stream can turn from a pure liquid stream to a gas stream due to slugging slugs of liquid can be followed by gas bobbles covering the entire pipe for longer periods of time, and the meter has to constantly interpret the fraction of what is flowing through the meter, and use the best suited method for calculating the flowrates of each phase. And the MPM meter installed on Alvheim have two mode which it can jump between in a fraction of a second, if the fluid stream suddenly becomes a gas stream and can operate in both a wet-gas mode and a multiphase mode. Where the MPM meter detects only gas and switches to a wet gas measurement mode, or if it detects liquid operates in multiphase mode. The MPM flow meters can automatically detect and switch between operation modes

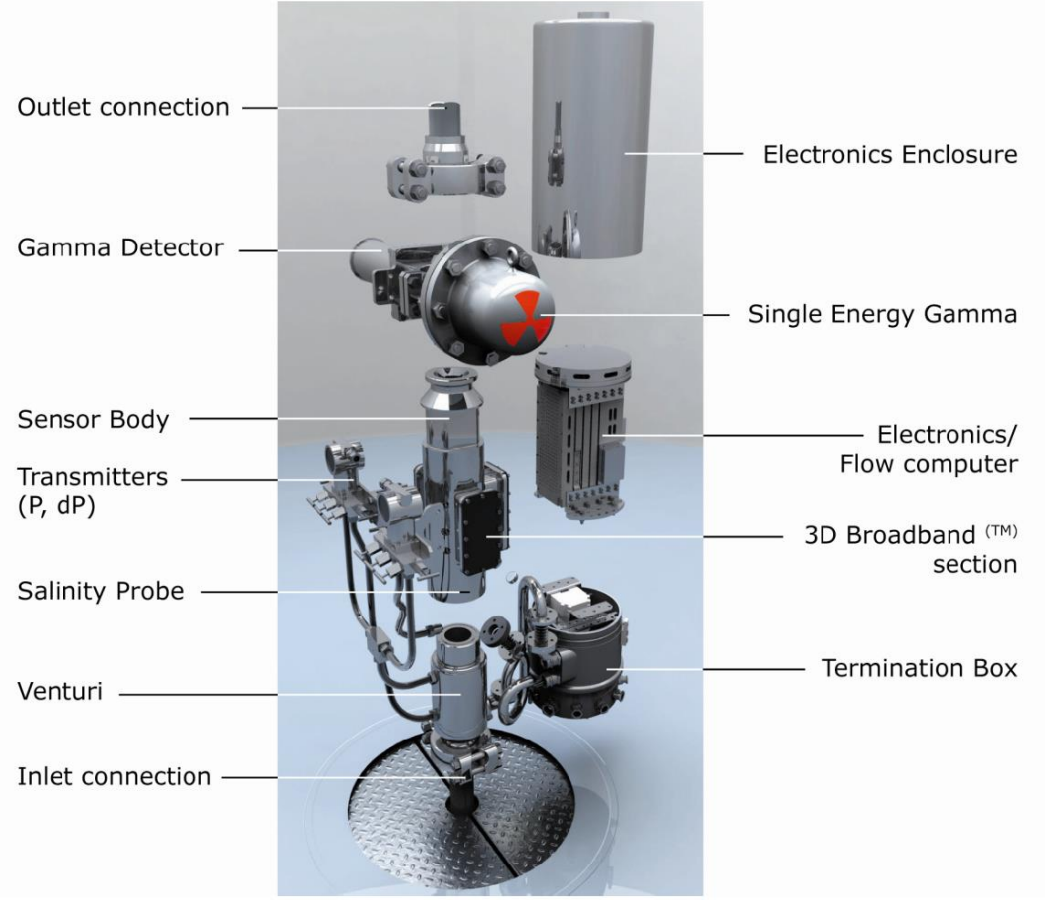


Figure 11 MPM Meter components [15]

4.5.1 Density measurement

In order to measure the mass density of media in the pipe the MPM multiphase flow meter uses a nuclear measurement principle, a source of ionizing radiation on one side of the stream and a Geiger muller tube on the other side, with a beam of photons / ionizing radiation going through the multiphase stream. The more photons counted by the Geiger muller tube the less mass the beam of photons has to permeate with a higher degree of being absorbed or deflected, and if the photon count on the Geiger muller tube decrees the mass density of the flow stream then has a higher mass density.

4.5.2 Velocity measurement of fluids

Most multiphase flow meters utilize a venturi flow meter to gauge the kinematic aspect of the multiphase meter, and measure a velocity component of the multiphase stream, and together with density measurement this can give good data of the velocities of the media in the conduit.

The established standards for using venturi flow meter, are mainly for single phase meters, so a special measurement model which has to take all this into account and still get a velocity estimate for the gas and liquid phases.

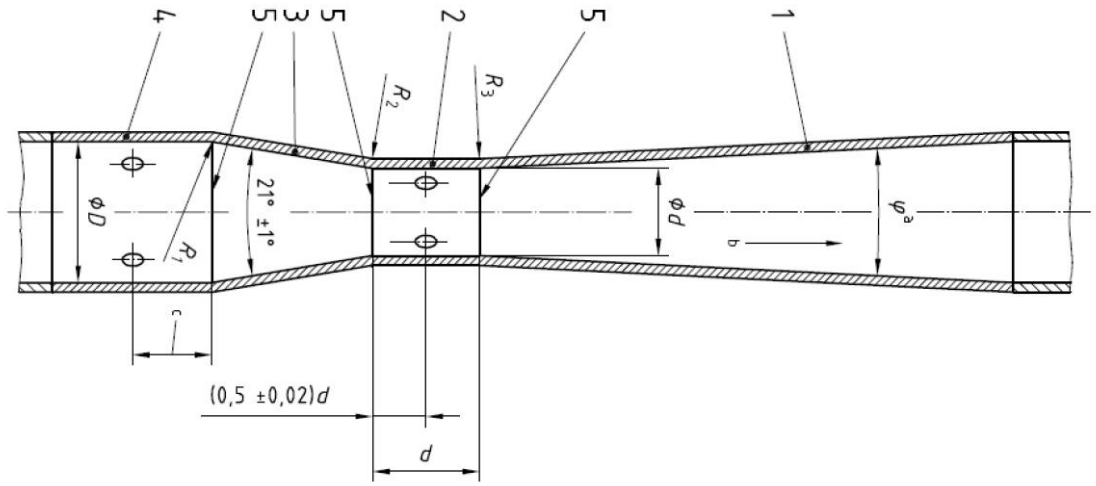


Figure 12 - Venturi cone element in a MPM meter [15]

4.5.3 Tomographic measurement

Through multi-modal parametric tomographic measure, a volumetric rendering of the cross-sectional area of the meter body can be measured, which intern can determine the ratio of gas and liquid within the meter body. This is done through what MPM called 3D Broadband™. This is gauged by multifrequency dielectric measurements in a similar manner as the water cut meter in the single phase oil stream the water content is inferred through a characteristic attenuation of electromagnetic waves, the MPM meter does this in a multi-modal approach by having multiple electrodes to measure the signal attenuation through the media in the pipe, and covers a frequency range of 20-3700Mhz. [17], as shown in Figure 13.

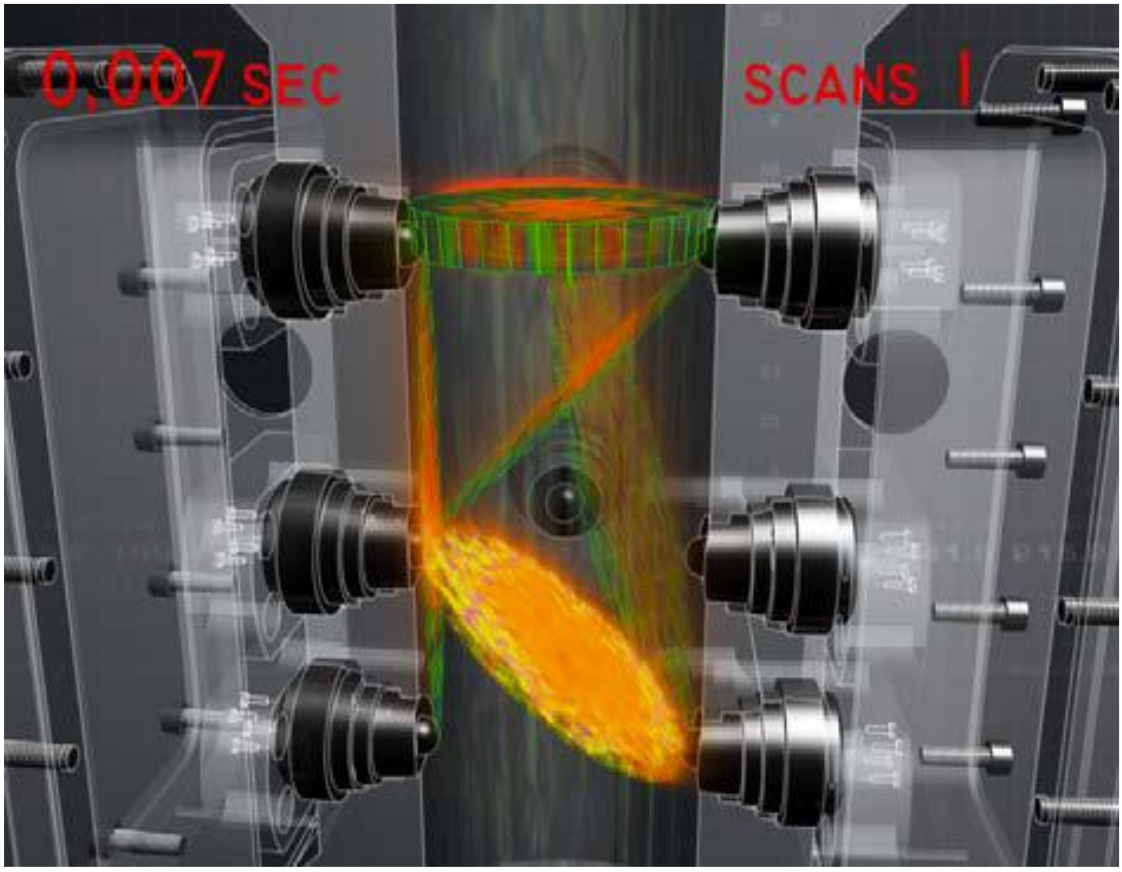


Figure 13 – Technip FMC- MPM 3D Broadband™ technology [15]

4.6 Operations / Maintenance of measurement equipment and systems

Under normal operations buildup of material inside the meter body can occur due to scaling, sulphates or asphaltenes this can effect can be take into account when a new calibration is performed, which is the biggest issue for the liquid and the multiphase meter, due to the abrasive nature of liquids and that the liquid is the dirtier of the two phases and that the liquid carry potential contaminants to a higher degree than gas, but in gas streams there are potentials for ice and hydrate formation.

4.7 Uncertainty

By carrying out a series of measurement points statistical analysis can be done on these measurements and as written in the calibration and traceability chapter under liquid flow meter some information about the use of small sample statistics through the student-t distribution is mentioned. And this is often a good place to get information about the repeatability, precision and uncertainty of the measurements and given a required confidence degree / coverage factor the stability of the same measurement can be achieved.

But when looking at expanded uncertainty of the combined measurement for single and multiphase metering the NFOGM have created uncertainty handbooks and programs to

calculate the uncertainties for both single phase gas and oil as well as a handbook for multiphase flow metering covering the topic to a great extent [18], but all uncertainties follow the *Guide to the expression of uncertainty in measurement* (GUM)[9]

4.7.1 Traceability

Traceability is the relationship a measurement has to an established international standard, and that each part of the chain from the instrument in duty to the standard has a documented relationship to internationally recognized standards.

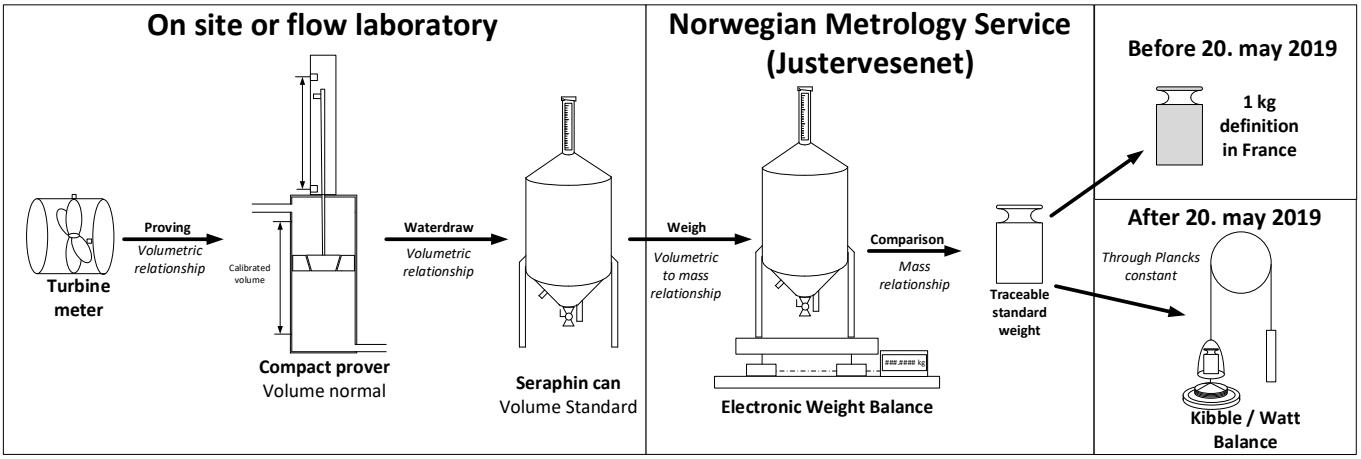


Figure 14 - Traceability map of a turbine flow meter

Figure 14 shows a traceability map example of a turbine flow meter and how it relates to the standards in France, from the 20th May 2019 the references are no longer physical objects but natural constants, such as lightspeed in a vacuum for a length, through interferometry. And the mass is based on Planck's constant through watt balance instrument [19] [20]

4.8 Supervisory metering system

The metering system has a Supervisory Control and Data Acquisition (SCADA) system running as a part of the measurement system which monitors, controls and creates reports of the measurement system. This also entails performing calibrations operating the metering stations and is done centrally on a main machine or with a redundant pair of machines. Each flow computer / embedded device is connected together, and central computer/s and these central machines run the programs and services, this central machine also performs as an interface to other SCADA systems or as data gateway for collecting or storing real-time or historical values on an ERP-level normally through OPC interfaces.

When looking at the data flow for the data used in this thesis as well as the interconnectivity of data across a typical metering system is shown in Figure 15, this shows some crucial aspects of data flow from the instrument through infrastructure until it is available for request for the processing done by this master thesis.

[9]The GUM is created by the Joint Committee for Guides in Metrology (JGCM) which Bureau International des Poids et Mesures (BIPM) shares. And that the international standard organizations such as ISO and IEC follow.

4.8.1 Data connectivity and interfaces.

Previously known as Object Link Embedding (OLE) for process control, but now it is just Open Process Control, and is a standard client/server networking protocol to transport data between components in process control and monitoring applications. And it is split in to two generations, the first generation which have the Data Access (DA), Historical Data Access (HAD) and Alarm and Events (AE) and some others which are based on Microsoft proprietary Distributed Component Object Model (DCOM), which require a network tunnel system for adding encryption. In recent years a newer generation of the OPC standard with more open standard based on open binary standard that don't require proprietary parts such as DCOM, which has benefits when it comes to encryption and security and implementation on multiple operating systems[10] this is called OPC Unified Architecture (UA). But for this thesis the old OPC DA is only used, which are in used between the Metering SCADA system and the Process control system and the historical value repository PI by OSIsoft. The Cognite Data Fusion repository collects the time series sensor data from this PI system. Figure 15 shows the data flow from both a single-phase stream measurement and a multiphase meter through the metering SCADA and the Operational Technology (OT) layer to the Cognite data fusion repository.

[10] For more details regards to these standards see https://opcfoundation.org/

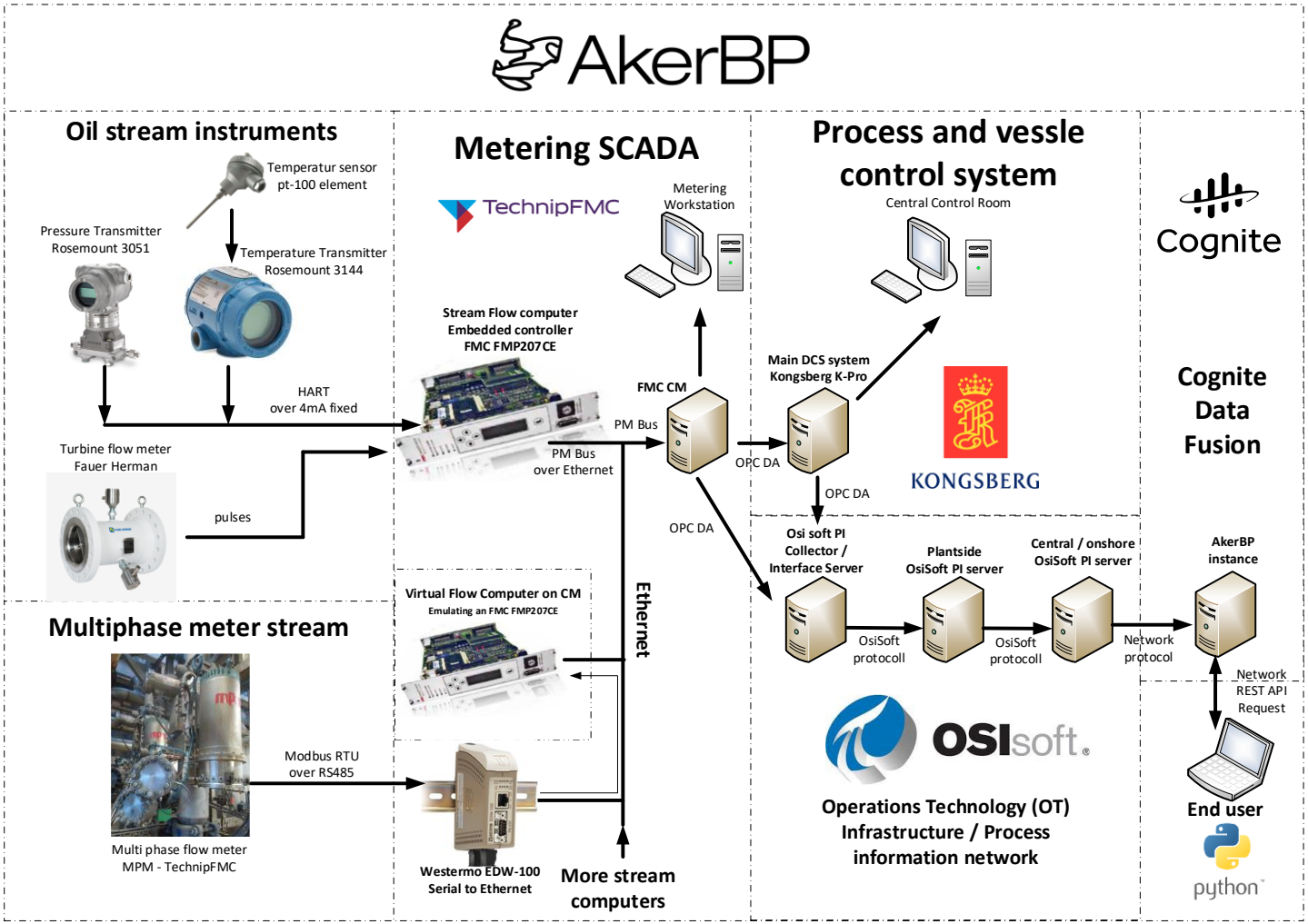


Figure 15 - Data flow for data used in thesis

5 Alvheim third party field installation

The chapter goes into detail of a third-party tie-in installation on the Alvheim FPSO, and the involved equipment and systems, and the documentation of the measurements system from the supplier is in appendix B

5.1 Alvheim field in general

Alvheim is a floating production storage and offloading (FPSO) vessel rebuilt from a shuttle tanker Odin to the Alvheim FPSO in Haugesund Norway.

Alvheim was initially developed for the Boa, Kneler and the Kameleon field, but further exploration in the area reveled more fields in the area, and that is where the third-party fields Bøyla, Vilje and Volund come in to the picture, third party tie-ins to the Alvheim FPSO require ownership allocation and separation, and this is where the topside multiphase meters on each stream and the third party separator comes into play. Figure 16 shows an overview of the metering system with regards to the Alvheim third party MPFM streams and separator, for executing single meter calibration.

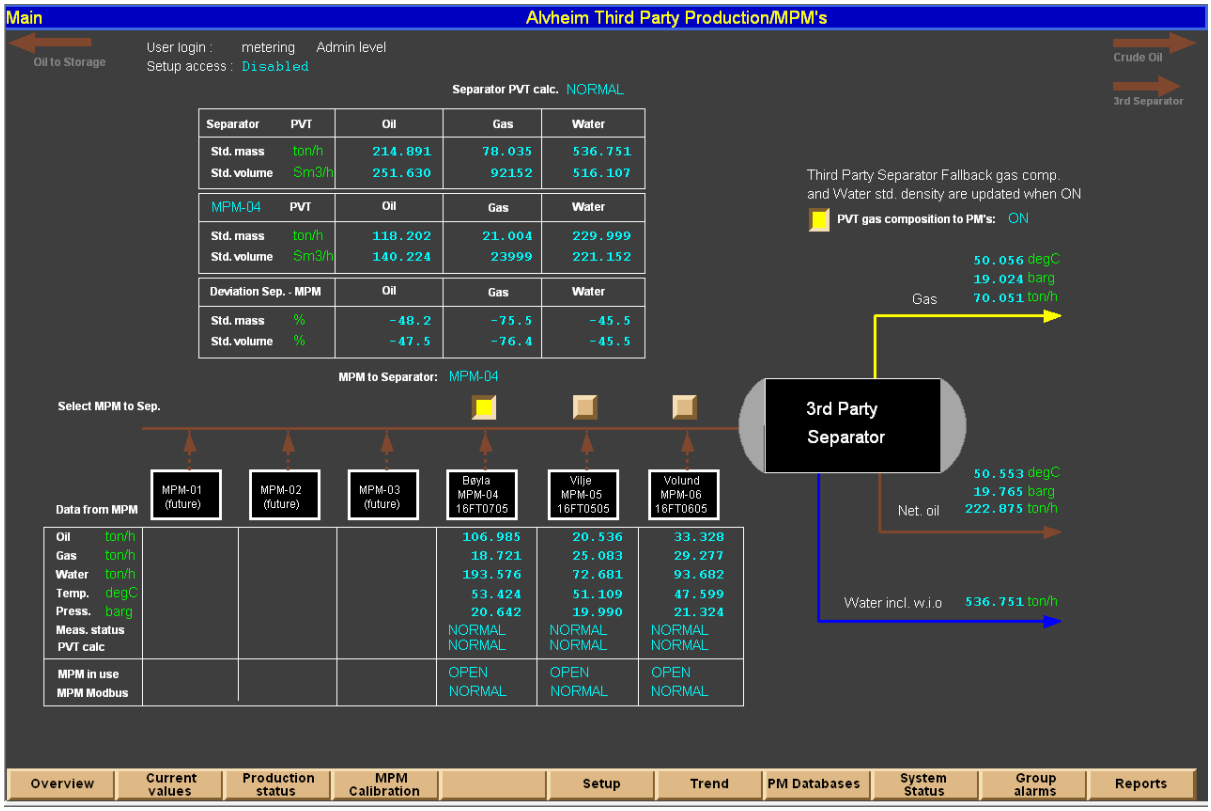


Figure 16 - Alvheim third party production and MPM's on Metering SCADA

5.2 Topside MPFM Manifold

The Topside MPFM Manifold on Alvheim has slots available for 6 MPFM’s and today there is installed 3 MPFM meters. Each multiphase meter has its own virtual stream flow

computer, and communicates between them initially from the meter over Modbus RTU on a differential signaling bus (RS485) which is then converted to ethernet and made available for the virtual flow computer running on the central metering machine (CM) as shown in Figure 15.

Figure 17 - Vilje and Volund MPM - MPFMs

5.3 Third Party Separator

The Third-party separator is a vessel where gravimetric liquid separation of the well streams from the third-party manifold as well as segregating gas from the liquid phase in addition to flashing / evaporation of a fraction of the liquid phase goes to gas. There are instruments on the levels of the oil and water phase, as well as the pressure and temperature in the separator is measured, monitored and controlled. The produced water stream is then measured through a electromagnetic flow meter, The oil and gas streams have additional instrumentation and will be covered more in sub-chapters. An image of the separator is shown in Figure 18.

Figure 18 - Alvheim 3rd party separator

5.3.1 Oil stream

The oil metering on the separators oil outlet consists of two 8” streams, where one of the streams can choose to either use a turbine meter or a ultrasonic flow meter (USFM). Upstream the flow meter streams there is a inline water cut meter as well as a fast loop side stream for density measurement and sampling, this is shown in the simplified P&ID in Figure 19 and a photograph of one on the oil streams with notation is show in Figure 20.

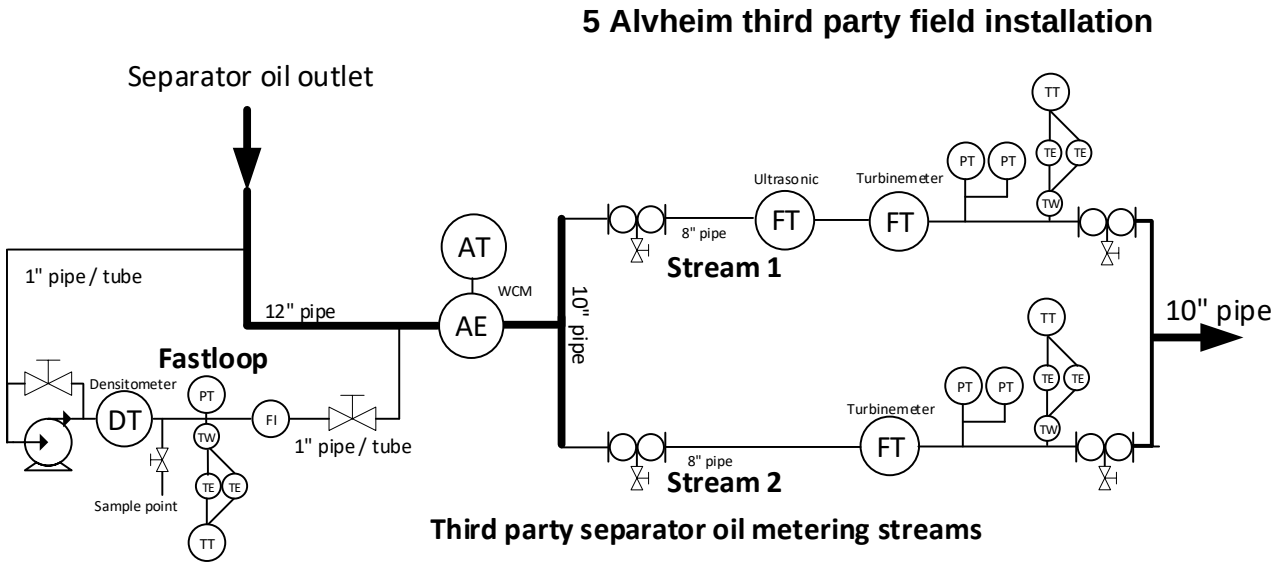


Figure 19 - Simplified P&ID over third-party separators oil metering station

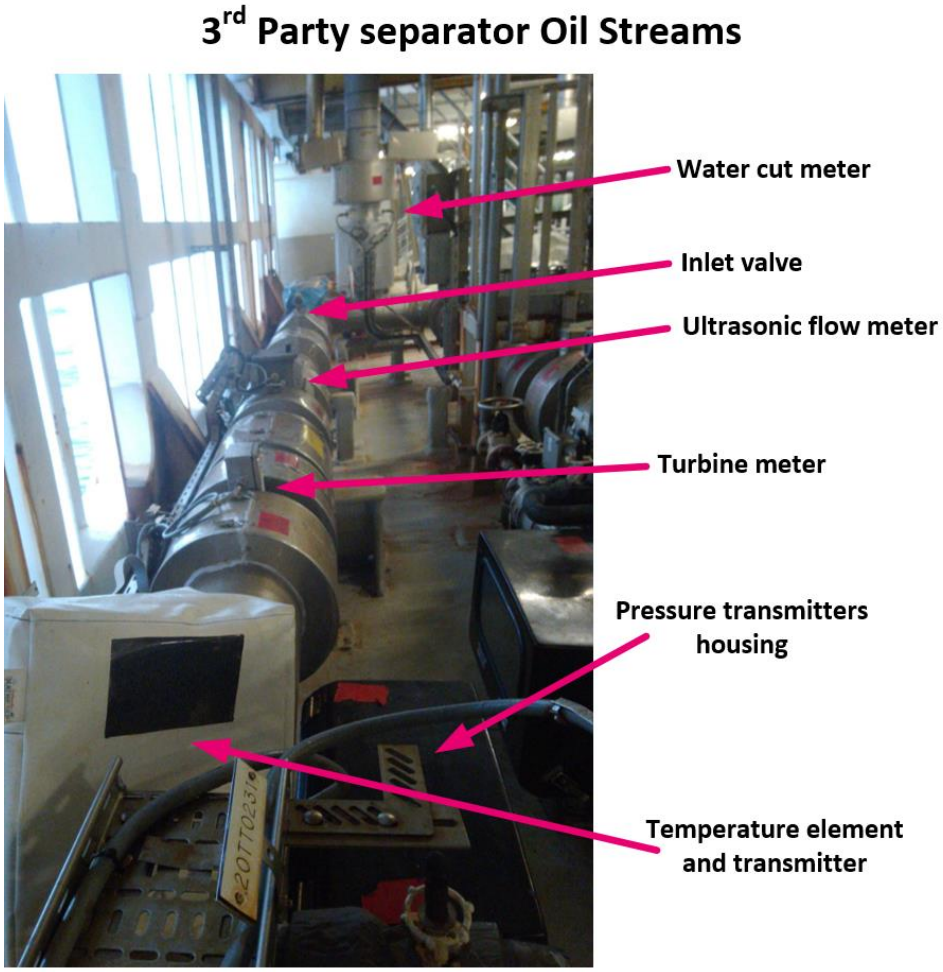


Figure 20 - picture of oil stream 1 with explanations

When it comes to the point where volumetric oil flow turns into mass flow the densitometer and the other equipment shown in Figure 21 comes into effect on the measurement. This is a 1” side stream with a representative oil flow through.

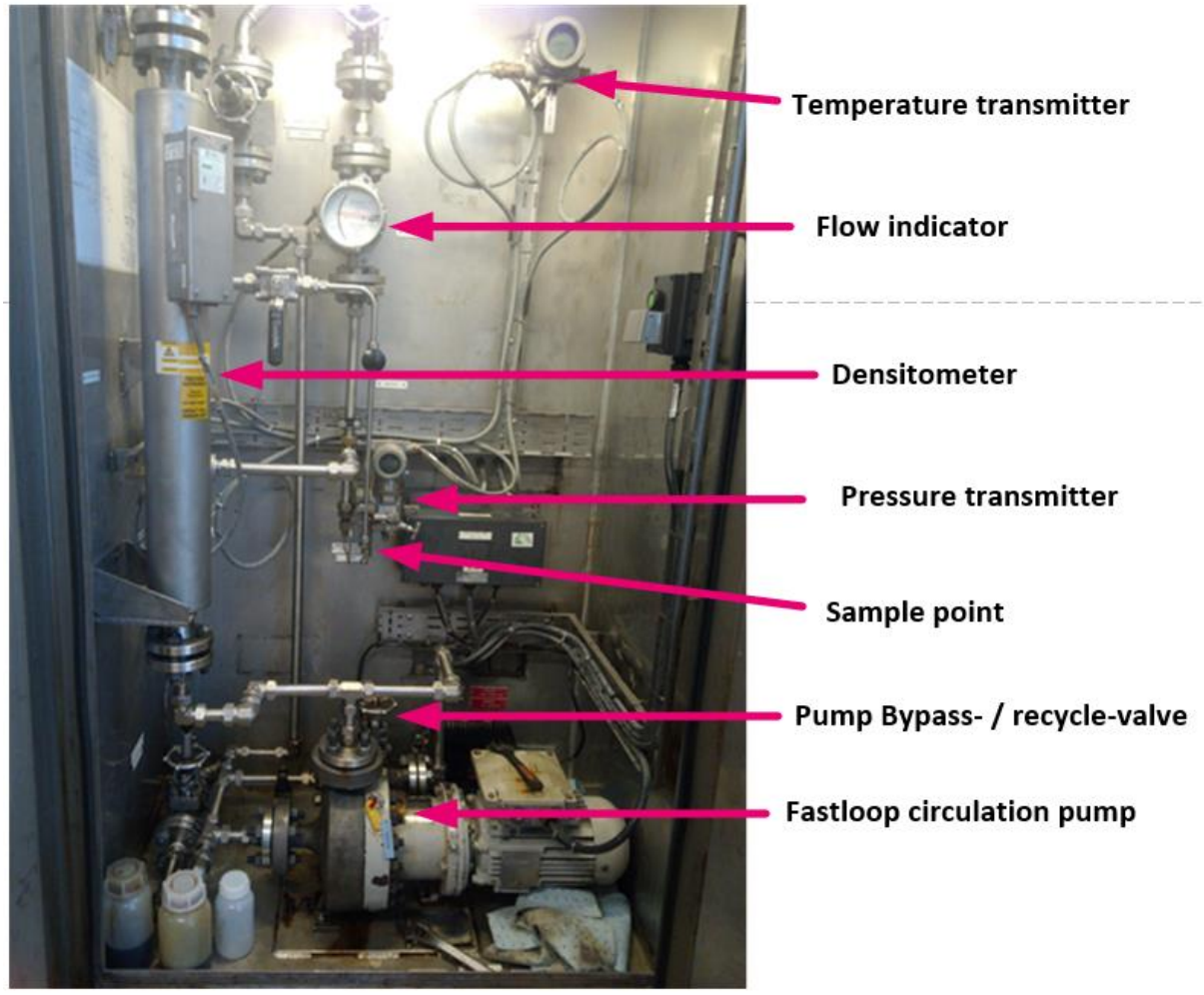


Figure 21 - oil stream fast-loop

5.3.2 Gas Stream

The gas metering on the separator’s gas outlet consist of two streams in parallel, same as the oil metering, but the gas streams have just one flow meter on each run, and both are ultrasonic gas flow meters, together with the same standard pressure and temperature instruments. This is shown in Figure 22.

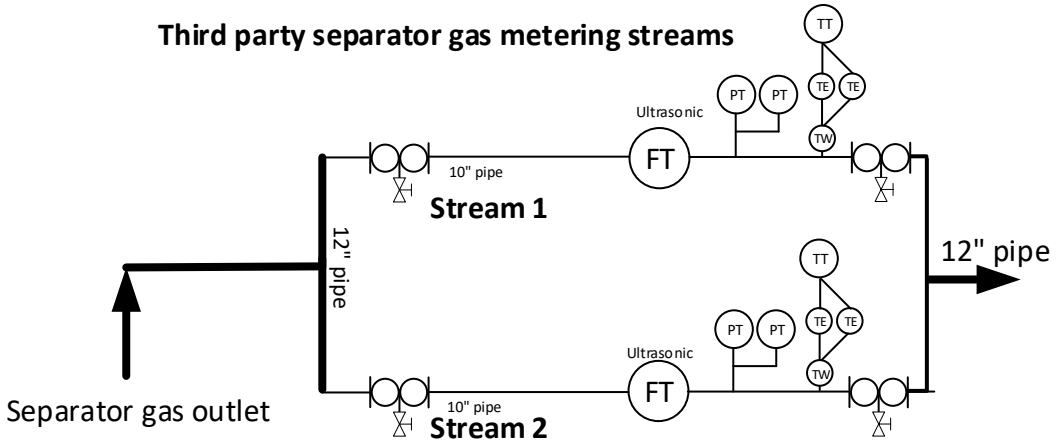


Figure 22 - Simplified P&ID over third-party separators gas metering streams

6 Parallel calibration of multiphase flow meters

This chapter goes into the essential parts of calibrating multiphase meters, and how a parallel calibration algorithm is created, implemented and executed.

6.1 Digital representation of fluid streams

In order to perform the calibration methods, the data used for the methods needs to be put in the correct context and calculated to fit the reality at any given instance. This is done by implementing a digital "twin" / digital representation of each of the streams in question. Both the multiphase streams of the MPFM's as well as the Separators effluent single-phase streams. And by developing a digital twin for these fluid streams which can take any arbitrary time segment, and perform calculations and accumulate the flowrates in each of the streams, a representable and highly adjustable data set is then created, and the calibration methods can then elegantly be fitted on top of a collections of these dataset calculated as calculated by the digital twin of the fluid streams. Figure 23 and Figure 24 shows the core concepts of the most important data within the digital representations, but in reality the digital representations stores also the related intensive variables and flowrates and performs the flashing calculations and prepare the data for further processing. Appendix C goes into detail of the implementation, development and calculations executed by these digital representations and how they are set into context.

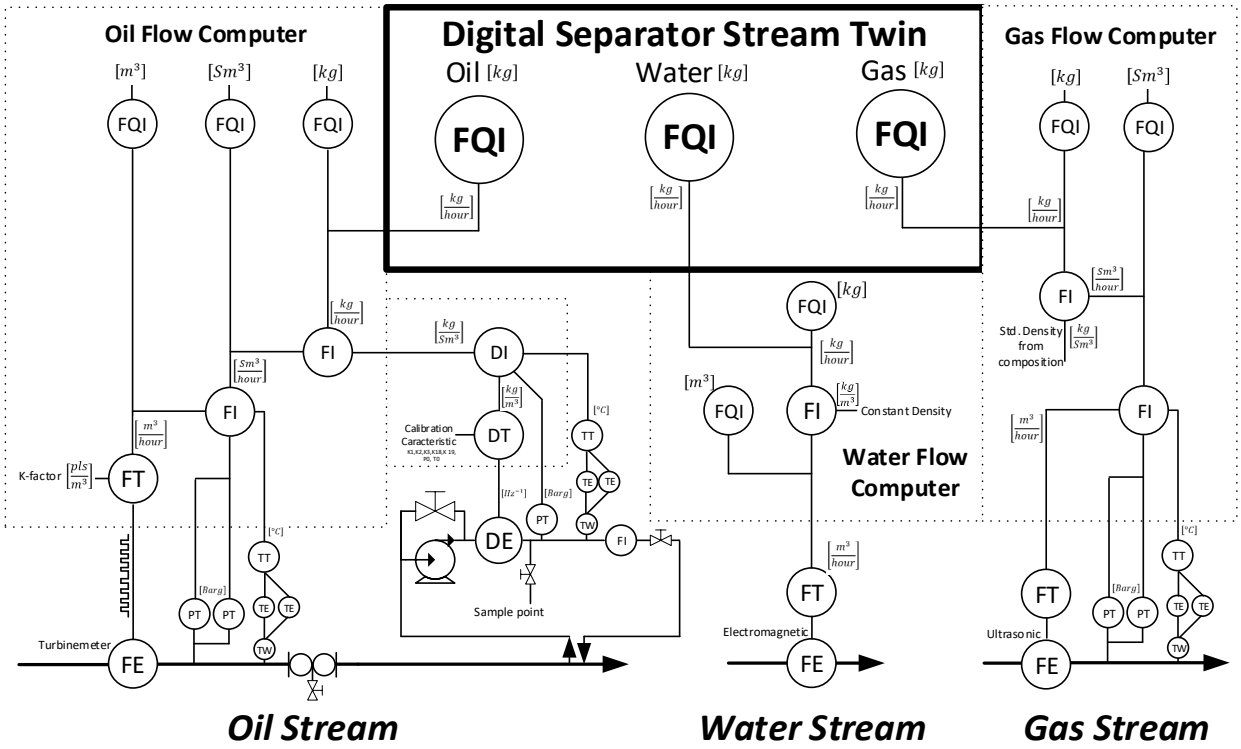


Figure 23 - Concept of Digital twin of separator streams

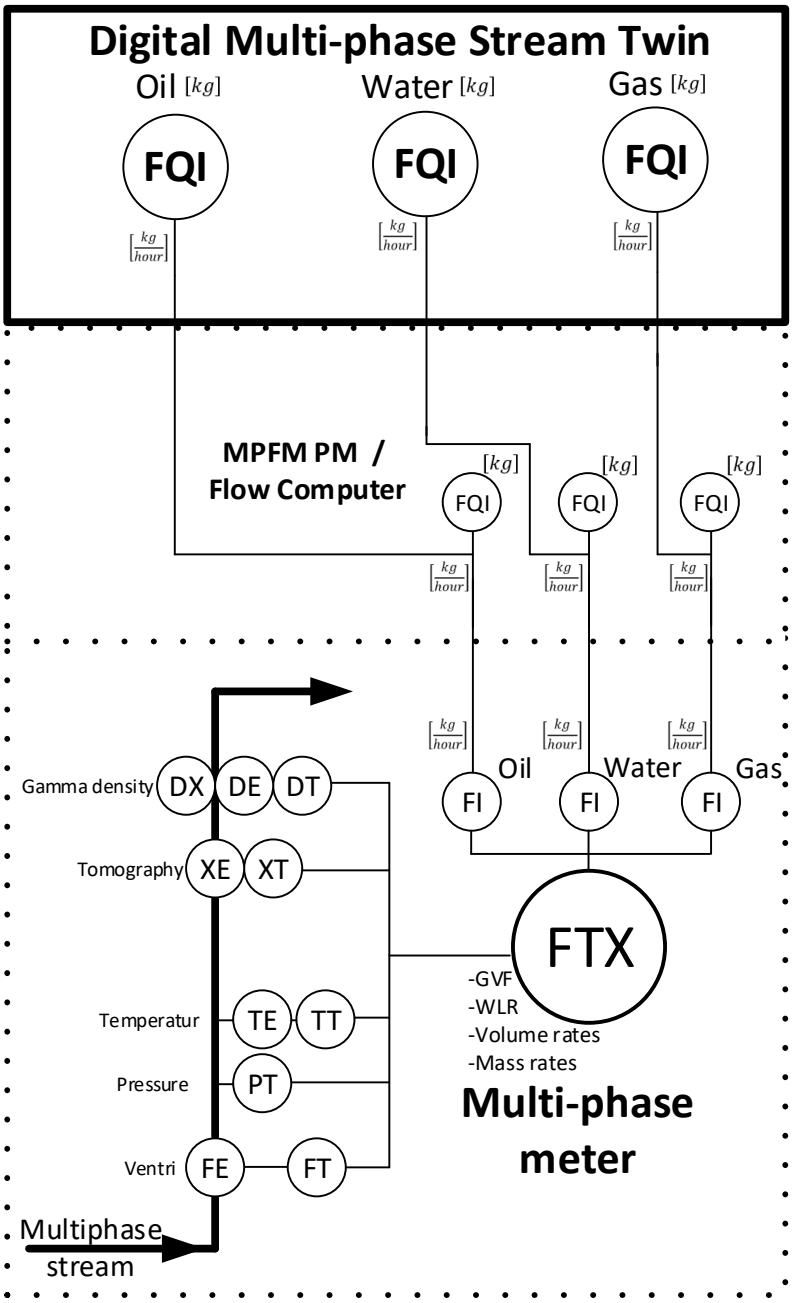


Figure 24 - Concept of Digital twin of a multiphase stream

6.2 Traditional calibration method

This chapter will cover the traditional MPFM vs separator calibration method used on the Alvheim FPSO for calibration of Allocation MPFMs with reference to separator measurements. And is only calibrating one meter at the time, and the meters on the outlets of the separators are the more accurate reference meters. And this Traditional calibration method will serve as a reference for the performance of the Parallel calibration method, but the Parallel calibration will use the same data but synthesize this into a parallel calibration task. The reason why the separator is used as the reference instrument is due to that single-

phase measurements have a much higher accuracy and repeatability and a lower uncertainty than the multiphase meter in the multiphase stream.

6.2.1 Calibration factor calculation

Figure 25 depicts the concept of the traditional method for calibrating where the effluent and influent accumulation of each phase are compared to each other.

The calibration factor k_p for each denoted phase p is calculated by the ratio between the accumulated mass flow of a reference measurement and the accumulated mass flow subject of the calibration subject, and over time this value as the accumulation of mass in both the reference and the subject this ratio practice stabilize / converge to a value which can be used as the calibration factor, as shown in equation (6.1). This accumulation is performed on all phases at the same time, from both the effluent single-phase reference streams on the separator and on the multiphase meter measuring all phases flowing through the meter. Then the calculation is performed on each of the phases to establish a calibration factor for each phase, during a calibration run[11].

$$k_p = \frac{\int_{t_0}^{t_1} \frac{dm_{p,ref}}{dt}\, dt}{\int_{t_0}^{t_1} \frac{dm_{p,sub}}{dt}\, dt} = \frac{m_{p,ref.}}{m_{p,sub.}} \tag{6.1}$$

[11] It is also worth noting that the factor is a fraction of mass divided by mass and hence a unitless number, and not a response factor turning a measured raw value to another unit like the calibration factor on a turbine meter which turns a number of pules into a volumetric quantity, and with the units in the factor

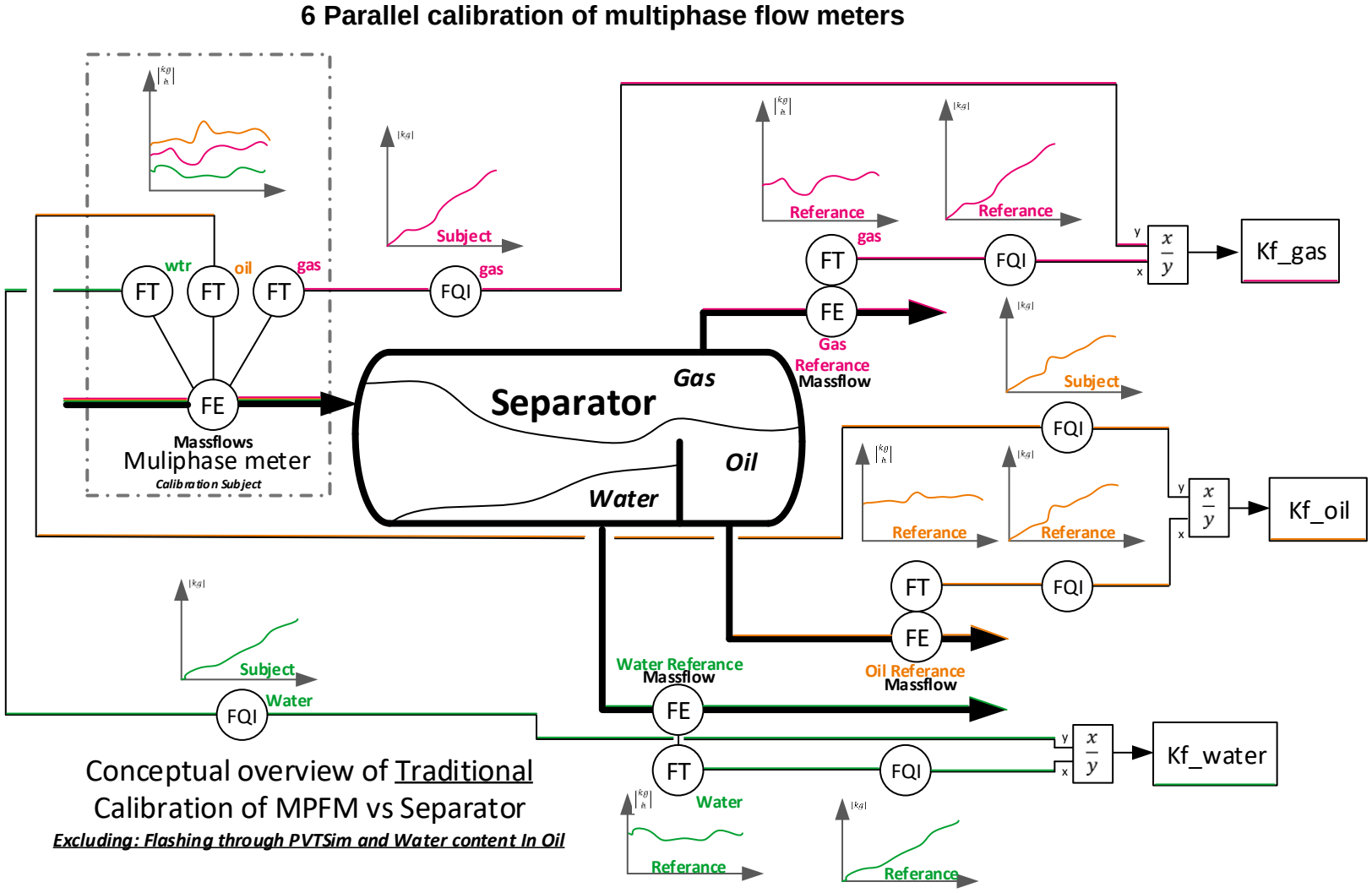


Figure 25 - Conceptual overview of traditional calibration method

Figure 26 - Not acceptable k-factor development through traditional calibration on the metering system on Alvheim

6.2.2 Acceptance criteria for a traditional calibration

The traditional calibrations accept criteria are qualitative to an extent and by looking at the calibration factors over the course of time of the calibration run. If the calibration factor for each phase remains constant over a longer period of time is the acceptance criteria of a calibration. Figure 26 shows a not acceptable calibration where the oil factor is not converging, Figure 27 shows on the other hand an acceptable traditional calibration run. And by being able to create / emulate these plots over time will give a good qualitative indication of the acceptability and convergence of the calibration run and will be a goal of the parallel calibration algorithm, to compare results to something known.

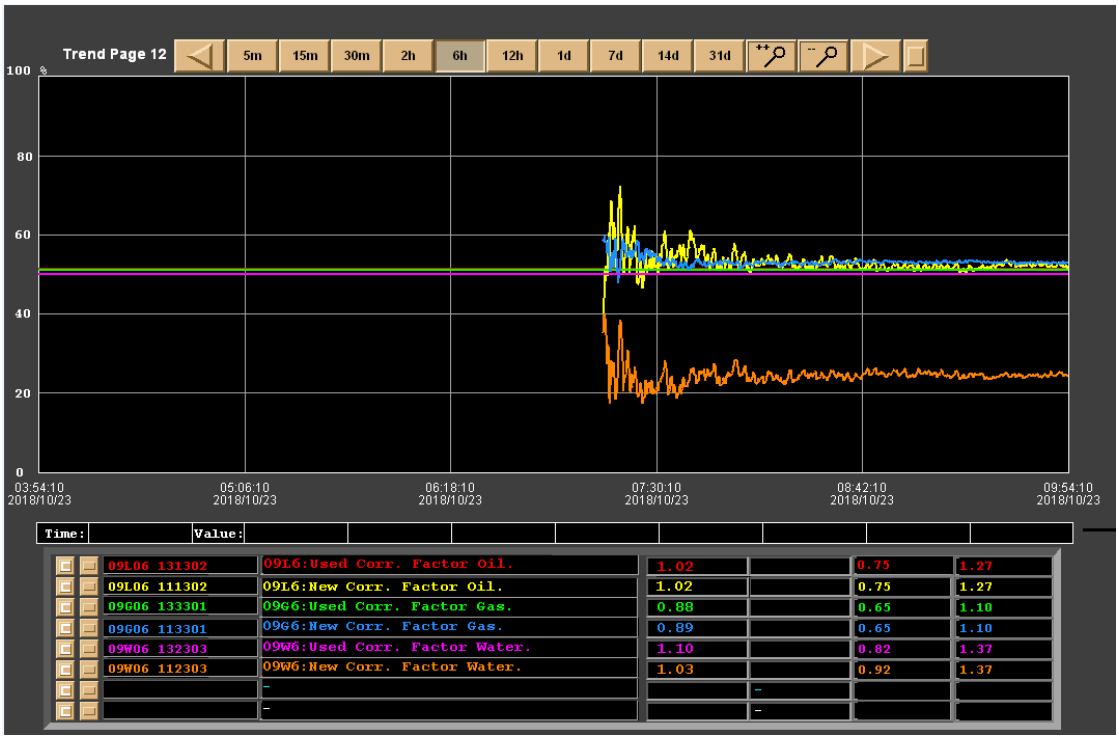


Figure 27 – Acceptable k-factor development through traditional calibration on the metering system on Alvheim

6.3 Parallel calibration method

This chapter will cover the theory, development and implementation of the parallel calibration method.

The general part of a parallel calibration routine / method it requires several different timeslots of historical data with varying flow rates in each of the streams named trials. And by performing a post calculation on these timeslots of historical data, achieve an apparent calibration factor. To achieve varying flow in each of the timeslots is done either by routing one of the streams to another separator, or a multiphase stream can be choked with a valve and they're by reducing the flow to a stream during a time period.

Two solvers have been looked into a linear and non-linear solver, where but the linear solver can also use all three runs at the same time, and not only have trials with only two streams on each trial as the non-linear solver. Figure 28 shows the dataflow and calculations done for the creation of each trial.

6.3.1 Calibration factor calculation

From the mass balance stated in 3.4.1 which stated that over enough time, the amount of mass entered the system must also leave the system. And that the multiphase mass flow is proportional to the single phase mass flow. In Alvheim current setup for the third-party installation with the topside multiphase meter streams. Bøyla, Vilje and Volund that are the multiphase influent streams, their rate must over time equate the effluent streams on separators single phase outlet, as shown phase generically in the equation below (6.2).

$$\underbrace{\left[\left(\int_{t_0}^{t_1}\frac{dm_{p,B\emptyset}}{dt}dt\right)k_{p,B\emptyset}\right]}_{\boldsymbol{B\emptyset yla}}+\underbrace{\left[\left(\int_{t_0}^{t_1}\frac{dm_{p,Vi}}{dt}dt\right)k_{p,Vi}\right]}_{\boldsymbol{Vilje}}+\underbrace{\left[\left(\int_{t_0}^{t_1}\frac{dm_{p,Vo}}{dt}dt\right)k_{p,Vo}\right]}_{\boldsymbol{Volund}}=\underbrace{\left[\int_{t_0}^{t_1}\frac{dm_{p,Sep}}{dt}dt\right]}_{\boldsymbol{3rd\ Party\ Separator}} \quad (6.2)$$

And by manipulation of the equations a system of equation for each phase can be set up and solved to achieve the calculated k factors for each stream and each phase in the multiphase streams.

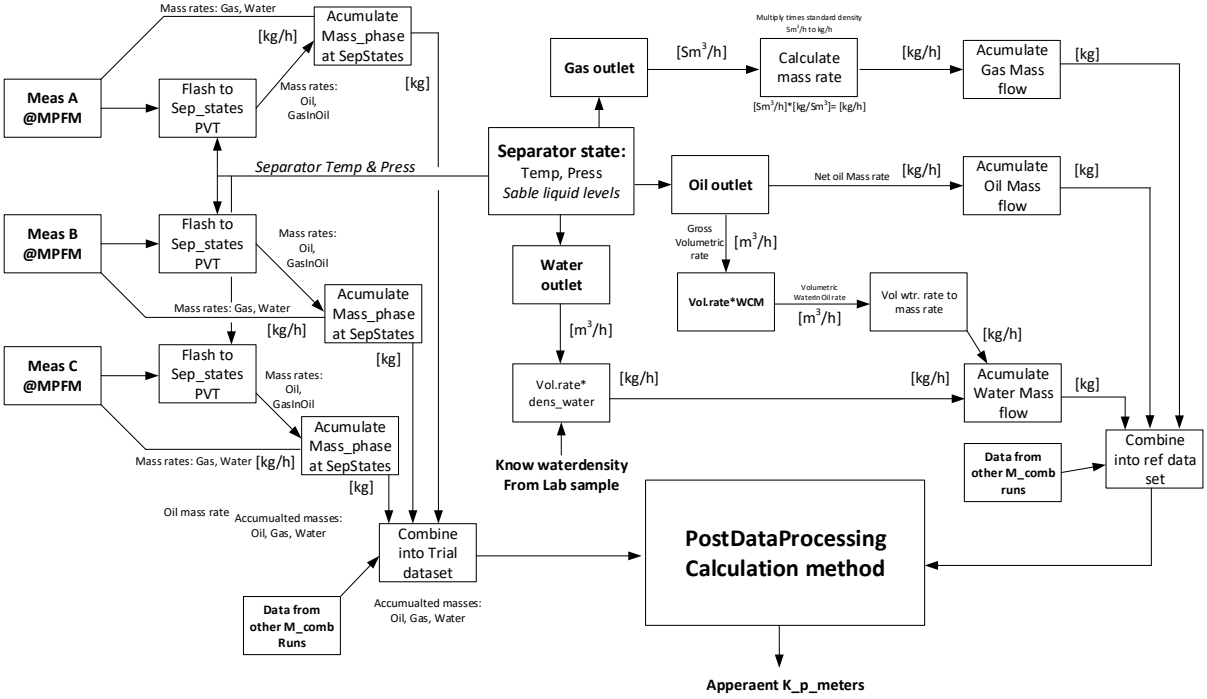


Figure 28 - Data basis, flow, calculations, and preparation for parallel calibration method solver

6.3.2 Non-linear solver

Initially there was a non-linear method proposed by Torbjørn Selanger but this was linearized and solve as a linear case, a chapter in Appendix D goes into detail of the method and the code for a solver, but the non-linear method was not implemented in the Parallel calibration algorithm, but it was solved.

6.3.3 Linear solver

The linear solver for the parallel calibration method, uses the same number of trials as there are parallel multiphase streams. And can be solved for as many phases as there is in the stream, it could also be used for single phase calibration.

6.3.3.1 Defining the specific case for the current Alvheim separator configuration

For the Alvheim third party parallel multiphase calibration method we have from the balance law stated (6.3) and by having three separate time window / trial post calculated for historical data. We can integrate the flow rates and be left with the accumulated mass pr phase times the k factor which sums up to the accumulated mass for the specific phase out form the separator. And from these three trial / time windows a system of equations can be set up as

show in (6.4). The system of equations have a linear relationship and the linear combination of the streams can be set up in the matrices and vectors shown in (6.5), which gives the linear system shown in (6.6) which can be solved by taking the inverse matrix of the stream trial matrix $\boldsymbol{M}_p$ and dot multiply it times the mass reference vector $\boldsymbol{m}_p$ which solves the system of equations for the unknown calibration factors $\boldsymbol{k}_p$ for each of the streams for the given phase (6.7). This is then repeated for each fluid phase.

$$m_{p,Bøyla} \cdot k_{p,Bøyla} + m_{p,Vilje} \cdot k_{p,Vilje} + m_{p,Volund} \cdot k_{p,Volund} = m_{p,ref} \tag{6.3}$$

$$\begin{aligned} m_{p,Bøyla,1} \cdot k_{p,Bøyla} + m_{p,Vilje,1} \cdot k_{p,Vilje} + m_{p,Volund,1} \cdot k_{p,Volund} &= m_{p,ref,1} \\ m_{p,Bøyla,2} \cdot k_{p,Bøyla} + m_{p,Vilje,2} \cdot k_{p,Vilje} + m_{p,Volund,2} \cdot k_{p,Volund} &= m_{p,ref,2} \\ m_{p,Bøyla,3} \cdot k_{p,Bøyla} + m_{p,Vilje,3} \cdot k_{p,Vilje} + m_{p,Volund,3} \cdot k_{p,Volund} &= m_{p,ref,3} \end{aligned} \tag{6.4}$$

$$\begin{matrix} \boldsymbol{M}_p & & \end{matrix} \quad \begin{matrix} \boldsymbol{k}_p \\ \end{matrix} \quad \begin{matrix} \boldsymbol{m}_p \\ \end{matrix}$$

$$\begin{bmatrix} m_{p,Bøyla,1} & m_{p,Vilje,1} & m_{p,Volund,1} \\ m_{p,Bøyla,2} & m_{p,Vilje,2} & m_{p,Volund,2} \\ m_{p,Bøyla,3} & m_{p,Vilje,3} & m_{p,Volund,3} \end{bmatrix} \begin{bmatrix} k_{p,Bøyla} \\ k_{p,Vilje} \\ k_{p,Volund} \end{bmatrix} = \begin{bmatrix} m_{p,ref,1} \\ m_{p,ref,2} \\ m_{p,ref,3} \end{bmatrix} \tag{6.5}$$

$$\boldsymbol{M}_p \boldsymbol{k}_p = \boldsymbol{m}_p \tag{6.6}$$

$$\boldsymbol{k}_p = \boldsymbol{M}^{-1}{}_p \boldsymbol{m}_p \tag{6.7}$$

6.3.3.2 Defining the general case

In the general case with as many multiphase streams collecting in our case into one reference separator / meter (parallel reference separators is also an opportunity for this method, just sum the same phase outlets)

But consider *S* number of multiphase streams, with *T* number of trials, where *S* and *T* are equal. and a single reference separator. And construct a matrix $\boldsymbol{M}_p$ of accumulated masses ($m_{phase,stream,trial}$) of multiphase stream, phase and trial (6.8). A reference vector $\boldsymbol{m}_p$ of the accumulated phases on the separator's outlets for each of the trials (6.9). (6.10) shows the vector $\mathbf{k}_\mathrm{p}$ of unknown k-factors in a specific phase for each of the multiphase meters. And by solving the linear system shown in (6.11) By taking the inverse of the $\boldsymbol{M}_p$ and dot multiplying it with the corresponding reference vector m_p the resulting $\mathbf{k_p}$ vector of apparent k-factors are found (6.12). And it is the same method for each of the phases.

$$\boldsymbol{M}_p \in \mathbb{R}^{T \times S} = \begin{bmatrix} m_{p,1,1} & m_{p,2,1} & \cdots & m_{p,S,1} \\ m_{p,1,2} & m_{p,2,2} & \cdots & m_{p,S,2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{p,1,\mathrm{T}} & m_{p,2,\mathrm{T}} & \cdots & m_{p,S,\mathrm{T}} \end{bmatrix} \tag{6.8}$$

$$\boldsymbol{m}_p \in \mathbb{R}^T = \begin{bmatrix} m_{p,ref,1} \\ m_{p,ref,2} \\ \vdots \\ m_{p,ref,T} \end{bmatrix} \tag{6.9}$$

$$\mathbf{k}_{\mathrm{p}} \in \mathbb{R}^{S} = \begin{bmatrix} k_{p,1} \\ k_{p,2} \\ \vdots \\ k_{p,S} \end{bmatrix} \tag{6.10}$$

$$\boldsymbol{M}_p \boldsymbol{k}_p = \boldsymbol{m}_p \tag{6.11}$$

$$\boldsymbol{k}_p = \boldsymbol{M}_p^{-1} \boldsymbol{m}_p \, , when \; S = T \tag{6.12}$$

6.3.3.3 Trials and the cumulative mass matrix

In order to perform the method Trial of streams with separate time windows and flow rates needs to be created. And each trial must contain a uniqueness to them either by rerouting a multiphase stream to another separator or have a variation in flow, either through well test or other effects, or physically created by choking the flowline through a choke valve, to create information to the measurement matrix $\boldsymbol{M}_p$. In order to solve the system of equation the measurement matrix $\boldsymbol{M}_p$ must be a square nonsingular / invertible matrix and have an inverse, and if all the trials building the matrix, must complete in a matrix with a rank *T*.

In essence the rank of the cumulative mass matrix must be equal to the number of Streams in parallel and hence the number of trials *T* equation (6.13), this is done by splitting each matrix row into different time slots called trials, and by ensuring uniqueness in each trial the matrix becomes invertible. And that $\boldsymbol{M}_p$ does not have linearly dependent rows.

$$\text{rank}(\boldsymbol{M}_p) = S = T \tag{6.13}$$

6.4 Parallel calibration algorithm

The algorithm solves the general case for *S* number of streams and utilizes the collections of trials, which is a list of the object orientation of the streams covered in Appendix C. These collections of data from each trial time window is synchronized an used to create the matrices and vectors for the method. All these synchronized datasets are assessed as runs, and these systems of equations are then solved for each synchronized run. And the resulting k-factors for each run contains the development of the k-factors across the different elapsed time within the timeslots. When these values are stabilized a subset of the stabilized values are used for a statistical analysis to assess the stability / random uncertainty of the calibration, in order to assess different k-factors. All this is explained in detail in appendix D on the parallel calibration algorithm. But in essence Figure 29 shows the essentials of the method solving inside the algorithm, but the part where datapoints are synchronized is left out of the figures, but the synchronization algorithm uses NumPy's core for efficient numerical computation, which is also mention in Appendix C and shown in Appendix D.

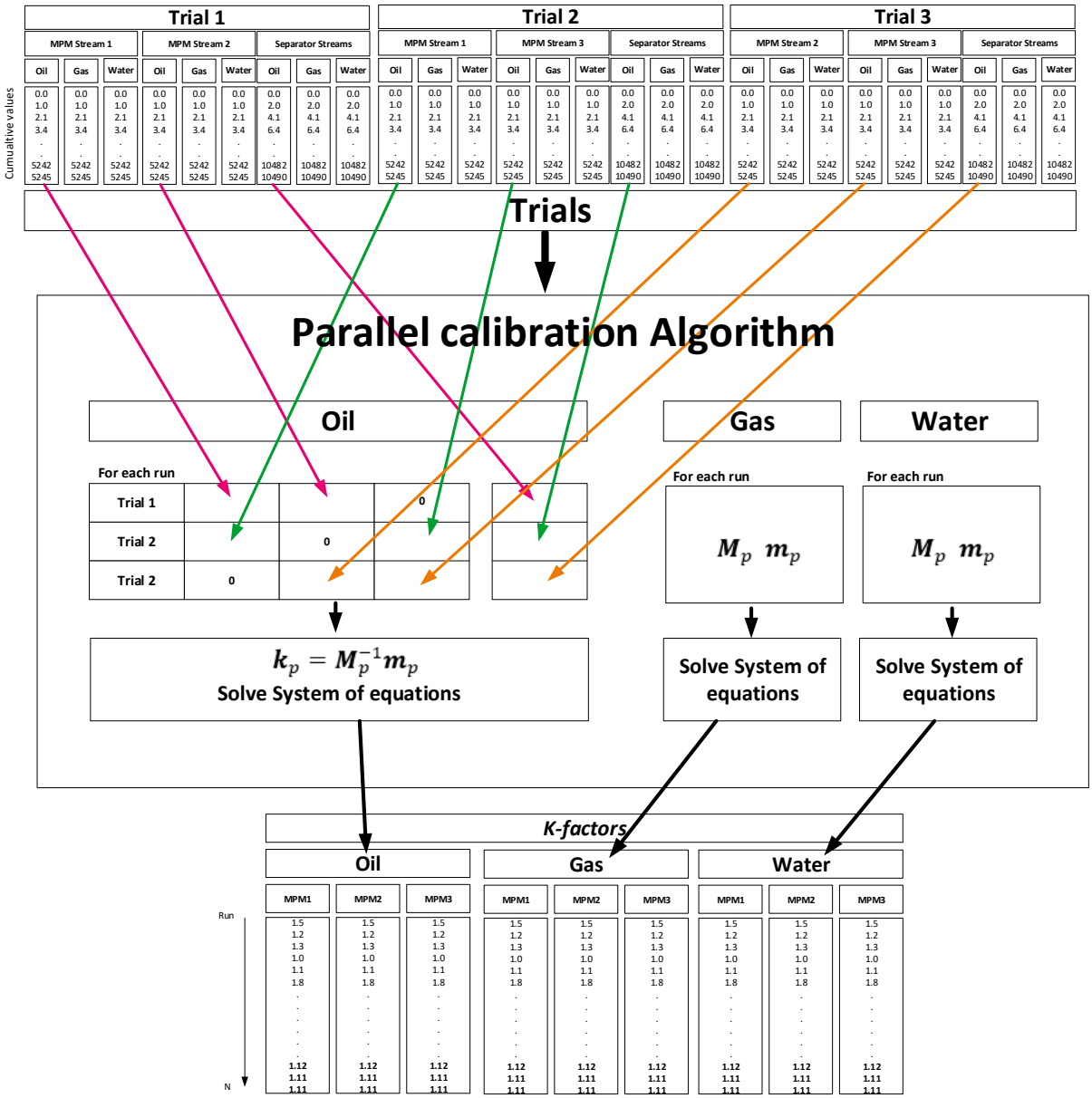


Figure 29 - Inside the algorithm - filling $\boldsymbol{M}_p$ Matrix and $\boldsymbol{m}_p$ vector with a 2x2x2 Trial input

6.4.1 Result and statistical analysis of the calibration method

When a parallel calibration is performed on the entire data set to provide the result of the calibration, the system can look into when the system has achieved a stable k-factor / converged to a stable k-factor, a window of samples of stable k-factor values called a statistical basis can be used for a statistical analysis of the basis of stabilized values. If there are no stabilized / converged values for a stream and phase the calibration is probably not successful.

The result of the k-factor for each phase and stream can be statistically defined as the mean, standard deviation and the number of sampled k-factors in the basis, with the random uncertainty of the data set as a numerical integration of the normal distribution over the confidence interval, coverage factor of 2, as specified by the measurement regulations [11]

Implemented in the algorithm the initial run of statistics executes a auto detector of a result basis for a statistical analysis, looking into the stability and random uncertainty of the

resulting k-factor, in order to create a initial basis dataset of k-factors as mention above. The basis is detected through peaks in squared differences of a squared sum of errors in the k-factor basis in a given stream, this is implemented as a preliminary, but a manual oversight of the selected statistical basis should be used and recalculated with a specified number of k-factors. Figure 30 show the result of a detected statistical basis with the k-factor development, with a histogram of the sample number with a fitted normal distributions density plot, in order to assess the resulting basis and that the values are normally distributed.

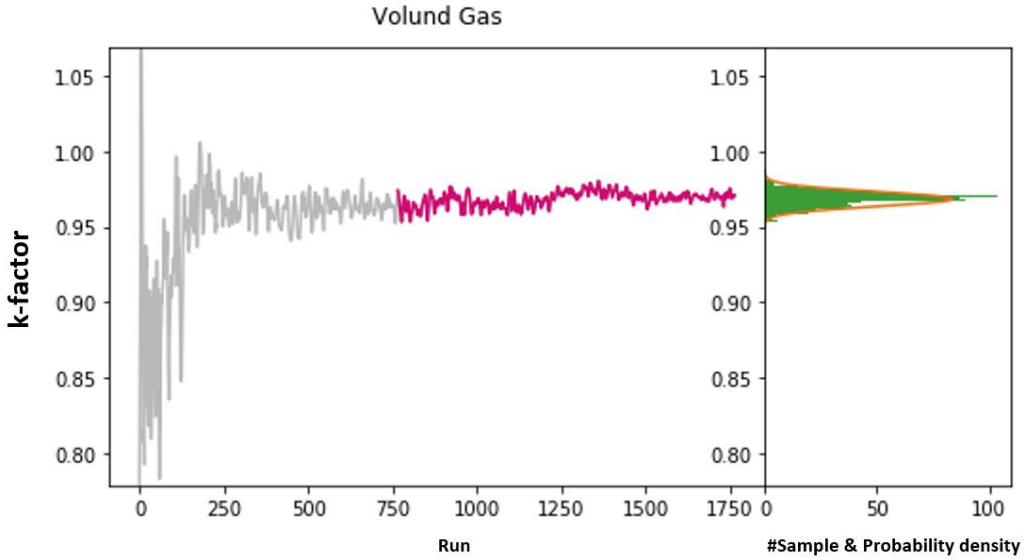


Figure 30 - Example of detected statistical basis (red) from k-factor development (gray), vertical samplel histogram (green) with a normal distrobution probability densityplot (orange)

6.4.2 Calibration evaluation

After the calculation of k-factors and the statistics on the stabilized values, a lot of visual tools for displaying the result and evaluation of both the raw and calculated values within the algorithm. An augmented matrix plot for different states is implemented, for both the flowrate, cumulative values and one for the essential intensive variables and the states within the separator. This gives an oversight of all essential values and context of the data within the algorithm. The augmented matrix plots a plot where each cell within the matrix contain a plot specific to the data within the stream and trial. Figure 31 show one of these augmented matrix plots created for ensuring the quality of the data, and in the separator conditions on the first trial, states of the oil level in the separator and the water cut in the separators oil stream is not optimal to assume constant mass in the system and stable conditions.

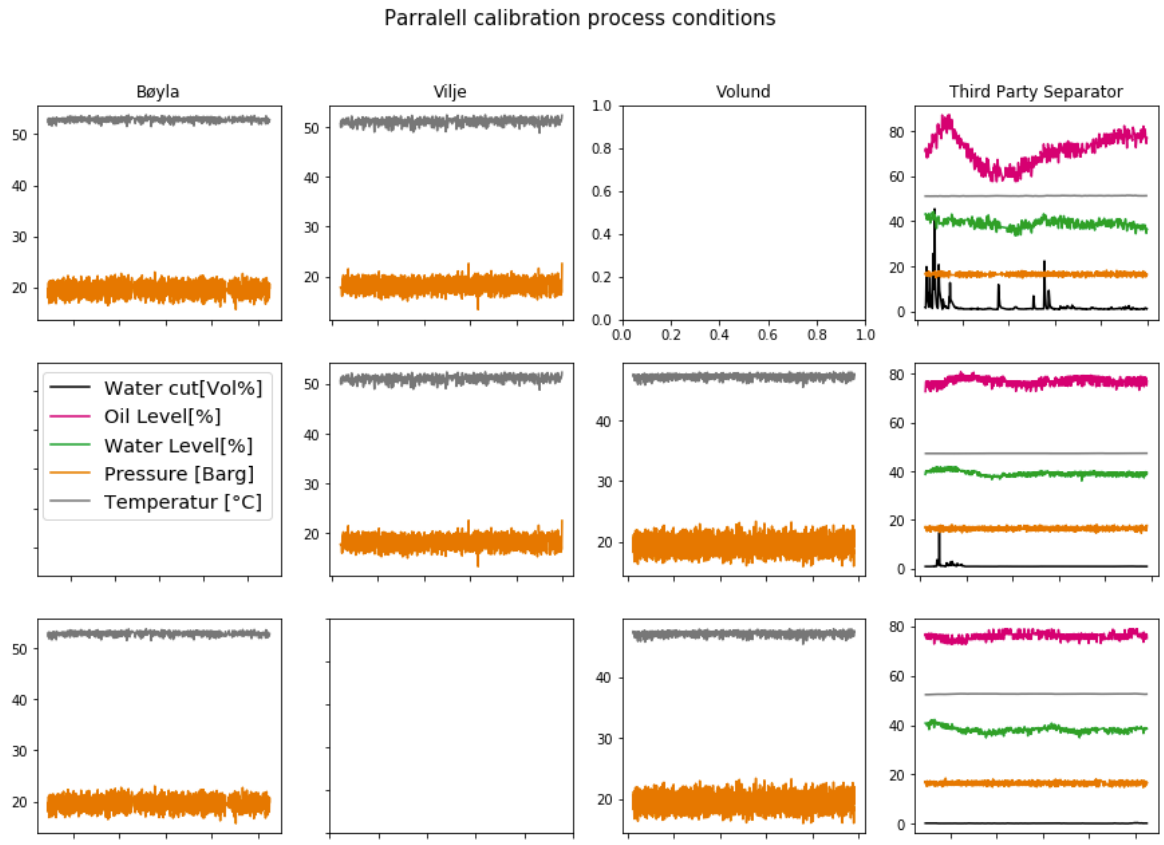


Figure 31 - Augmented matrix plot of Process conditions during trials, where the x axis is successive raw datapoint during the trial, and therefore no numbers.

6.4.3 Synthesizing data for comparing traditional calibration towards a parallel calibration

To be able to verify the parallel calibration method the result of a traditional calibration is used, but in order to perform a parallel calibration on the same data used in traditional calibration the one multiphase meter to one separator has to synthesized into a synthetic parallel trial dataset. By synchronizing and summing the separator streams to each other based on the elapsed time of the accumulation found in the accumulator class, then this can elegantly be used as a single trial. Perform the same summation and create the rest of the parallel trials and what is left is a synthetic dataset which can be used on a parallel calibration method.

6.5 Executing parallel calibrations

The parallel calibration algorithm is created to be executed in a interactive python session, such as IPython based environments such as Jupyter Notebook. A flowchart with a Jupyter notebook code shows the execution of a parallel calibration in Figure 32. Where the automatic trial window finder is used, but one trial is changed to cover a three subject streams in one trial, also note that the separator stream is the last stream appended to the list.

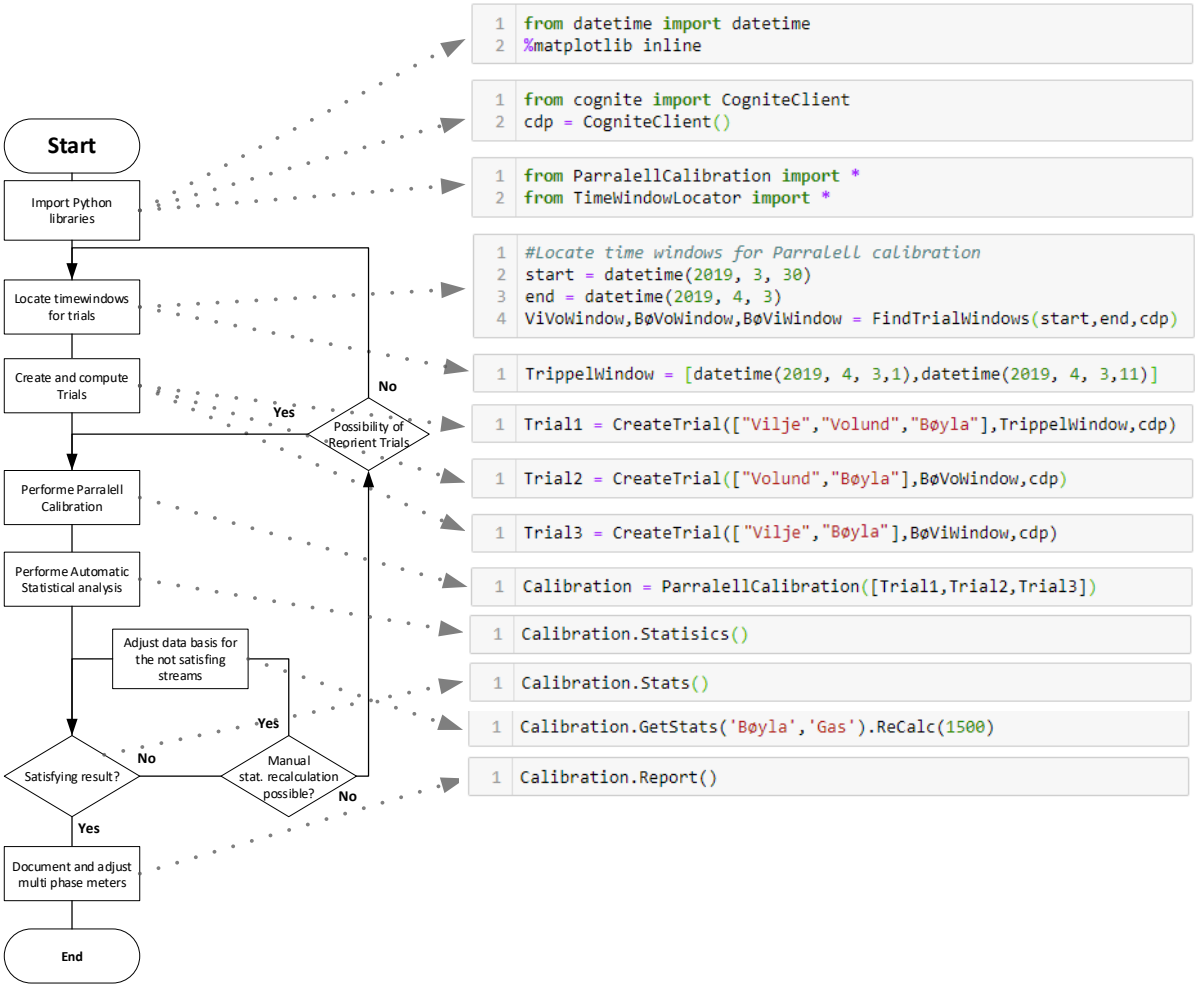


Figure 32 - Flow chart and method execution for a 3x2x2 calibration in an IPython environment

7 Results

This Chapter will go into the result of the calibration methods implemented and calculated both in the Traditional and Parallel method. When comparing the parallel calibration method towards a known excepted reference, the method used is then to synthetically combine traditional calibration sequences into a synthetic parallel calibration dataset. The parallel calibration then solves both the traditional data set and the synthetic and then compares this. During the time of this thesis there where two datasets of traditional calibration, one in late January and one in March and are covert in Appendix E on the January results and Appendix F on the April and some of the results will be shown below. And during the march and April test of parallel calibration was executed, giving real values and not just synthetic values as in the January 2019. The results from this chapter will be discussed in chapter 8.

7.1 January calibration

In January a normal calibration was performed on Alvheim and the resulting k-factor development of the January calibration is shown in Figure 33 for Bøyla, Figure 34 for Vilje and Figure 35 for Volund, comparing both developments of the traditional and the synthetic parallel dataset. And the numerically compared in Table 7.1 which also includes final values from real calibration.

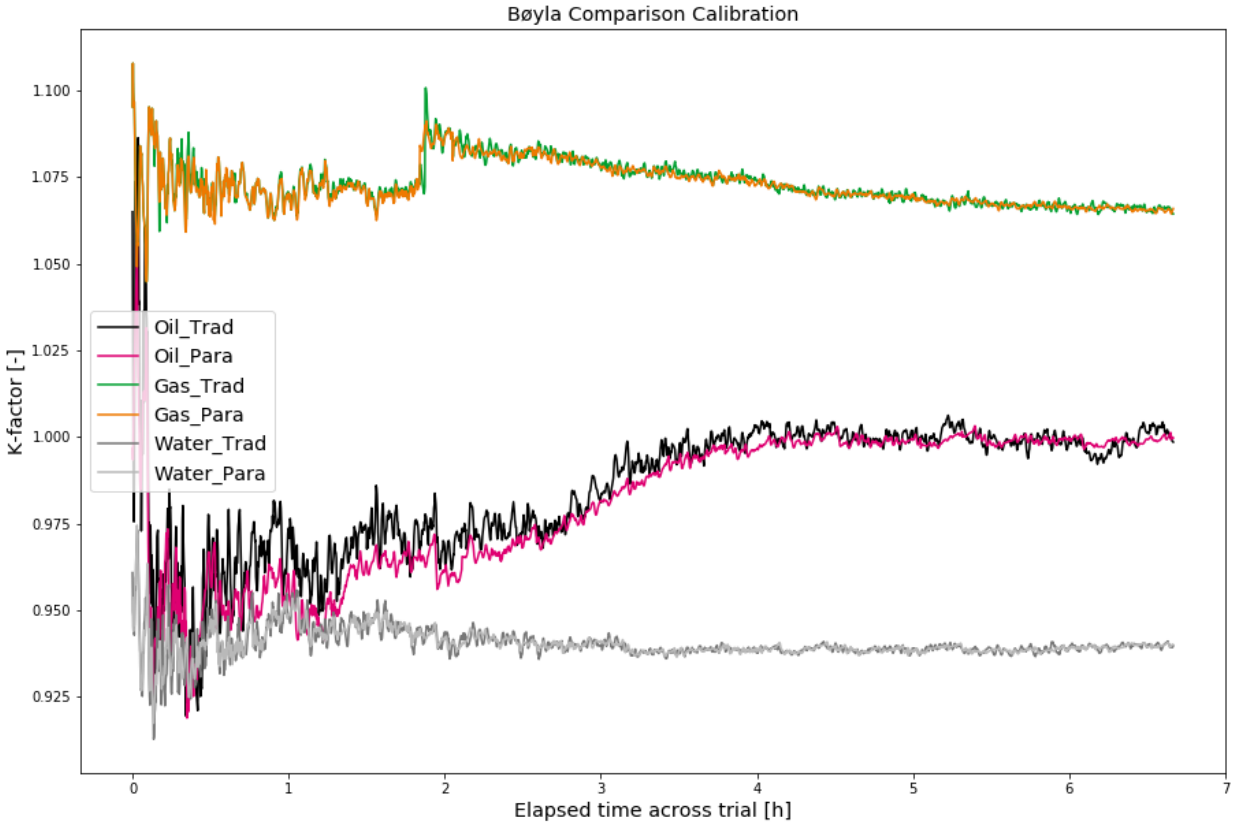


Figure 33 - k-factor development of Traditional vs Synthetic Parallel calibration for Bøyla in January 2019

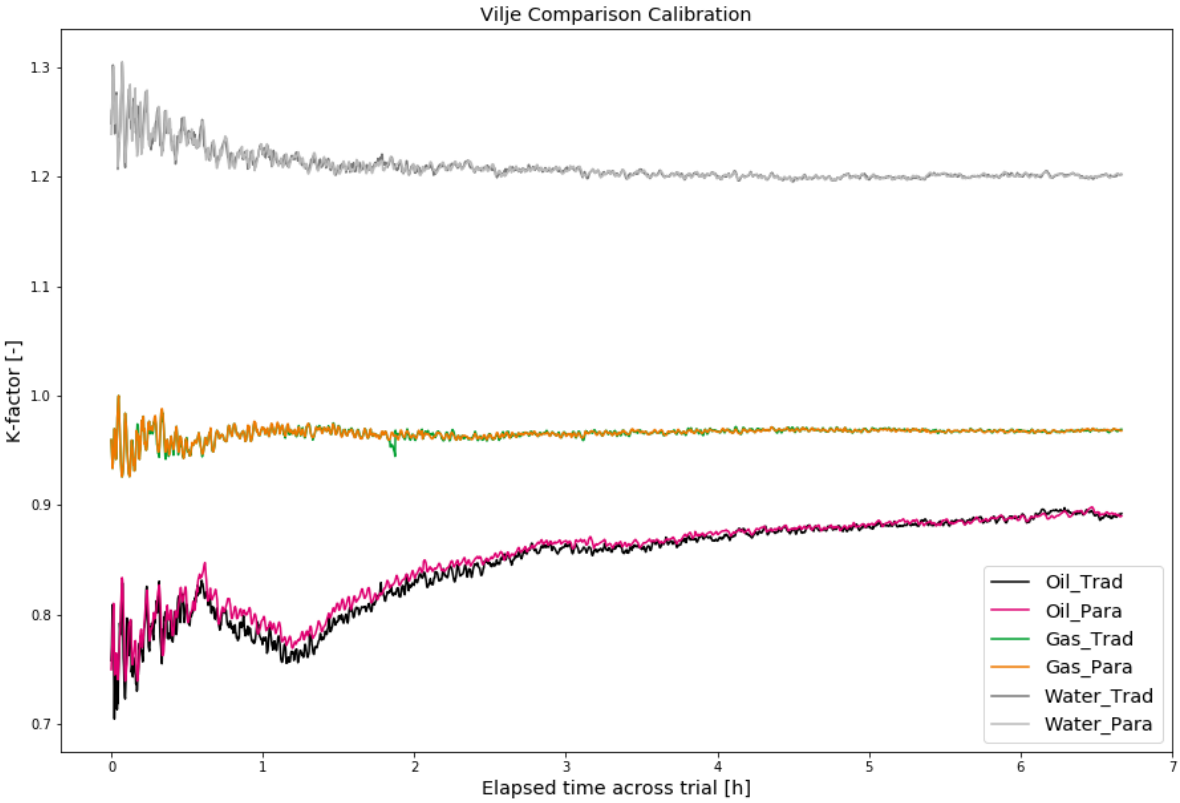


Figure 34 - k-factor development of Traditional vs Synthetic Parallel calibration for Vilje in January 2019

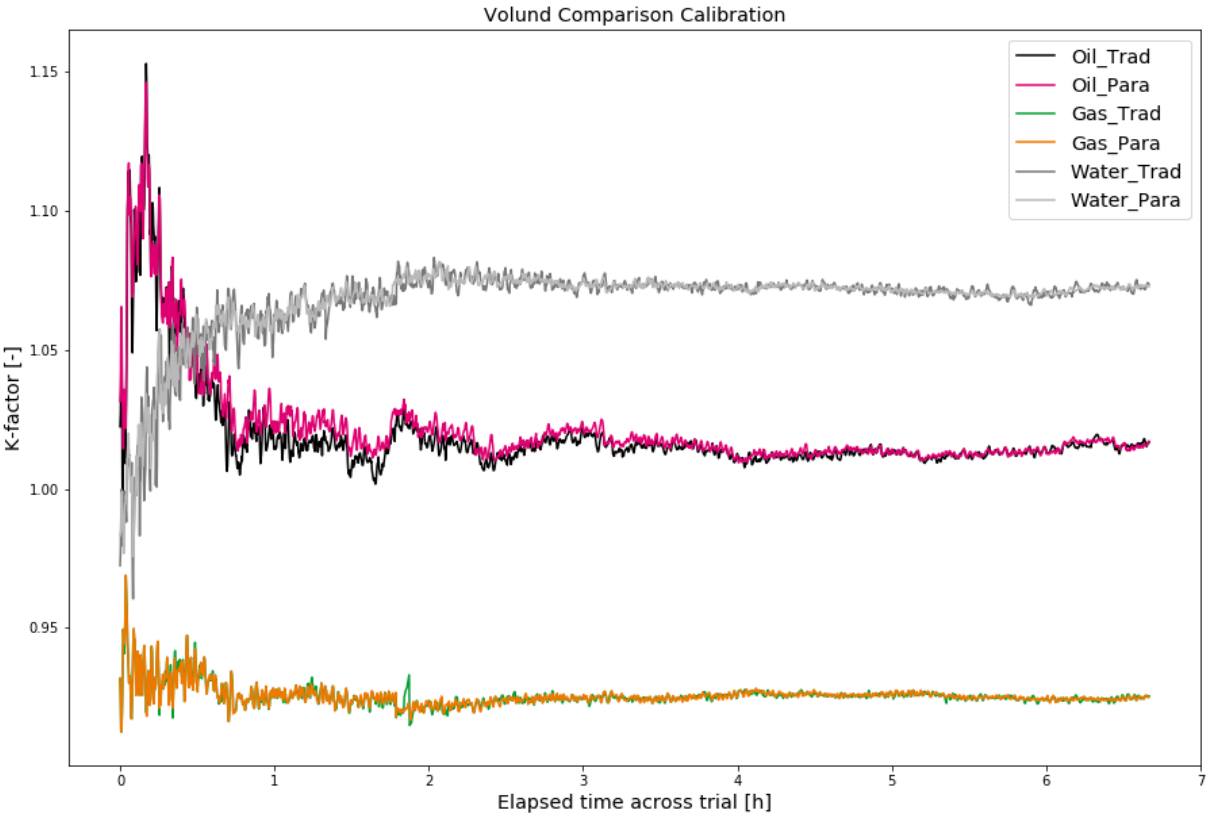


Figure 35- k-factor development of traditional vs synthetic Parallel calibration for Volund in January 2019

7 Results

Table 7.1: Resulting values of traditional and synthetic parallel calibration and the result form the existing fiscal metering calibration system in January 2019

January Calibration	Phase	Current metering system	Algorithm	
			Traditional	Synthetic parallel
Bøyla	Oil	0.97919	0.99874	0.99961
	Gas	1.01439	1.0666	1.06667
	Water	0.93353	0.93887	0.9391
Vilje	Oil	0.86164	0.88346	0.88274
	Gas	0.96043	0.9665	0.9665
	Water	1.15805	1.20046	1.20056
Volund	Oil	0.98797	1.01642	1.01431
	Gas	0.90674	0.92548	0.92536
	Water	1.04478	1.07209	1.07186

Table 7.2: Comparison between traditional and a synthetic 2x2x2 calibration by the calculated algorithm with differences and deviations

January Calibration		Traditional				Synthetic Parallel				Differences between Synthetic and Traditional (ref)			
Stream	Phase	mean	random uncertainty [%]	Standard deviation	sample size	mean	random uncertainty [%]	Standard deviation	sample size	deviation on k-factor	Deviation	diff rand. uncertainty	stability (St.dev-St.dev)
Bøyla	Oil	0.99874	0.99	1.42E-03	996	0.99961	0.54	2.59E-03	1060	8.64E-04	0.09%	-0.449	0.001
	Gas	1.06660	1.42	9.92E-04	500	1.06667	1.34	1.05E-03	500	7.00E-05	0.01%	-0.079	0.000
	Water	0.93887	1.83	7.69E-04	1000	0.93910	1.15	1.22E-03	1500	2.31E-04	0.02%	-0.673	0.000
Volund	Oil	1.01642	0.33	4.31E-03	1923	1.01431	0.41	3.40E-03	1947	-2.11E-03	-0.21%	0.088	-0.001
	Gas	0.92548	1.41	9.92E-04	1000	0.92536	1.42	9.92E-04	1000	-1.26E-04	-0.01%	0.001	0.000
	Water	1.07209	0.94	1.49E-03	1394	1.07186	0.83	1.70E-03	1327	-2.28E-04	-0.02%	-0.117	0.000
Vilje	Oil	0.88346	0.23	6.05E-03	958	0.88274	0.20	7.04E-03	935	-7.20E-04	-0.08%	-0.032	0.001
	Gas	0.96650	0.57	2.46E-03	1897	0.96650	0.49	2.86E-03	1897	1.00E-06	0.00%	-0.080	0.000
	Water	1.20046	0.78	1.79E-03	1122	1.20056	0.78	1.79E-03	1124	1.04E-04	0.01%	0.000	0.000

And compared to the real result from the existing system on Alvheim of the Volund results is show in Figure 36. And when it comes to the calculated uncertainty between a synthetic and a traditional calibration Table 7.2 goes int to the differences between the a traditional and a synthetic parallel calibration execution.

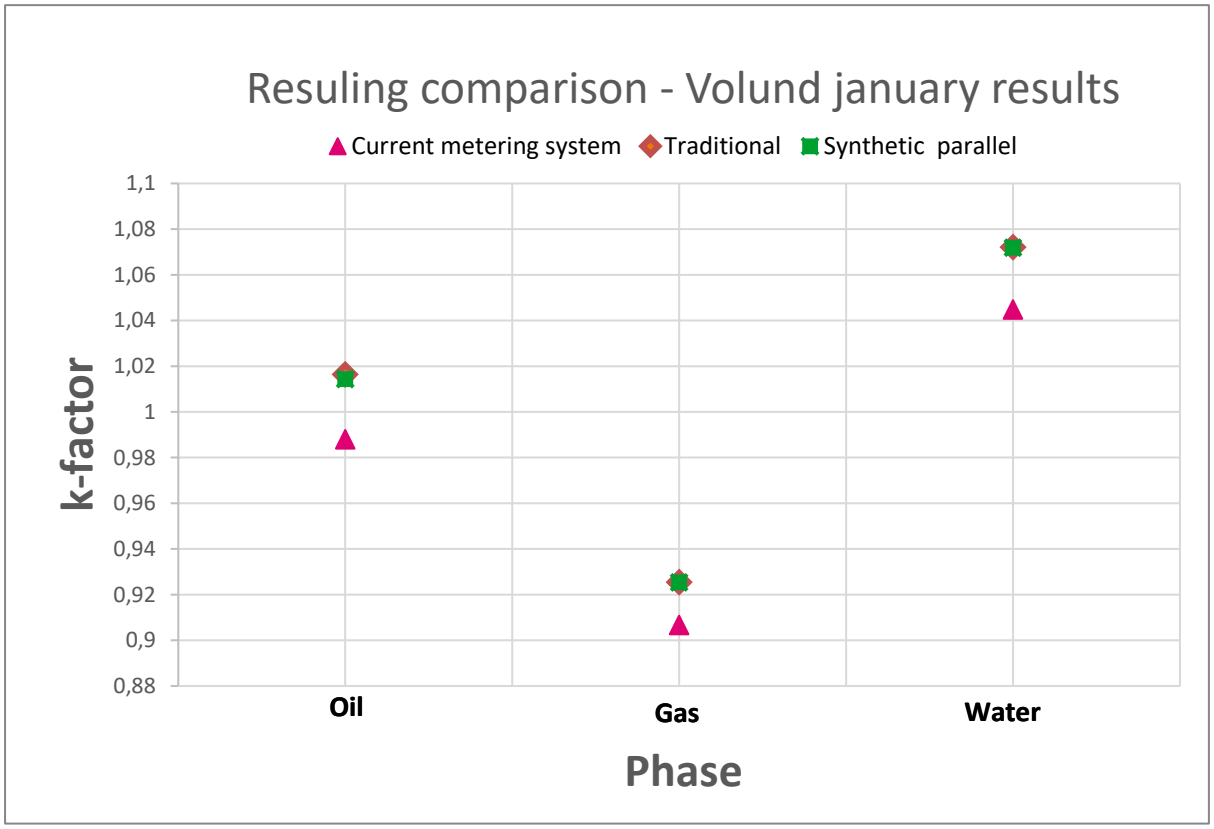


Figure 36 - Result comparison of Volund from January calibration trials, where the real value, traditional and a synthetic test was performed

7.2 April calibration results

An even better test from late March early April where performed, where first a traditional calibration was performed and after this a parallel calibration sequence was executed giving both a traditional test, with data for both a traditional run with a synthetic combination and in the days after the traditional calibration the parallel calibration could be performed keeping the traditional calibration as a benchmark for comparison. And then in late April a new parallel calibration was performed. Figure 37, Figure 38 and Figure 39 shows the results as point plots for each multiphase stream and is shown numerically in Table 7.3

Table 7.3: Resulting values of actual, and the algorithms ,traditional, synthetic parallel and parallel calibration in March / April and late April 2019

Calibration		Algorithm					
Time: March / April 2019		27/3 to 29/3			30/3 to 3/4		30/4 to 1/5
Stream	Phase	Current system	Traditional	Synthetic parallel	Parallel 2x2x2	Parallel 3x2x2	1mnd later 3x2x2
Bøyla	Oil	1.56405	1.59178	1.59117	1.55913	1.62318	1.51129
	Gas	0.90507	0.97910	0.97763	0.98940	1.05450	1.06952
	Water	1.07492	1.09826	1.09789	1.08120	1.00231	1.02029
Vilje	Oil	0.82050	0.86202	0.86305	0.78272	0.83145	1.11121
	Gas	0.97801	1.01513	1.01680	0.98864	1.06908	1.08579
	Water	1.13873	1.19756	1.19935	1.24515	1.14677	1.01351
Volund	Oil	1.14429	1.15180	1.15164	1.07059	1.03411	1.02004
	Gas	0.90674	0.94987	0.95168	0.97845	0.91330	0.96824
	Water	1.04478	0.98520	0.98450	0.98896	1.08478	0.99176

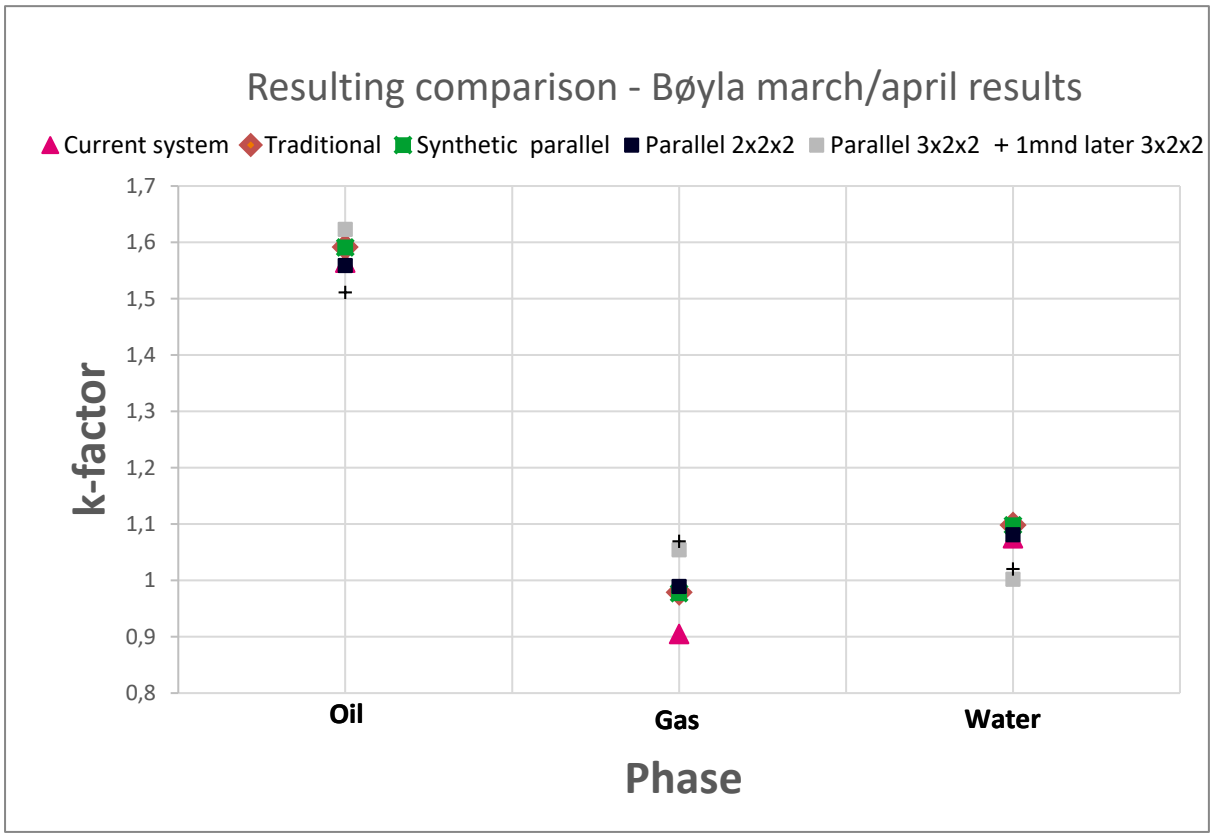


Figure 37 - Result comparison of Bøyla during March/April calibrations

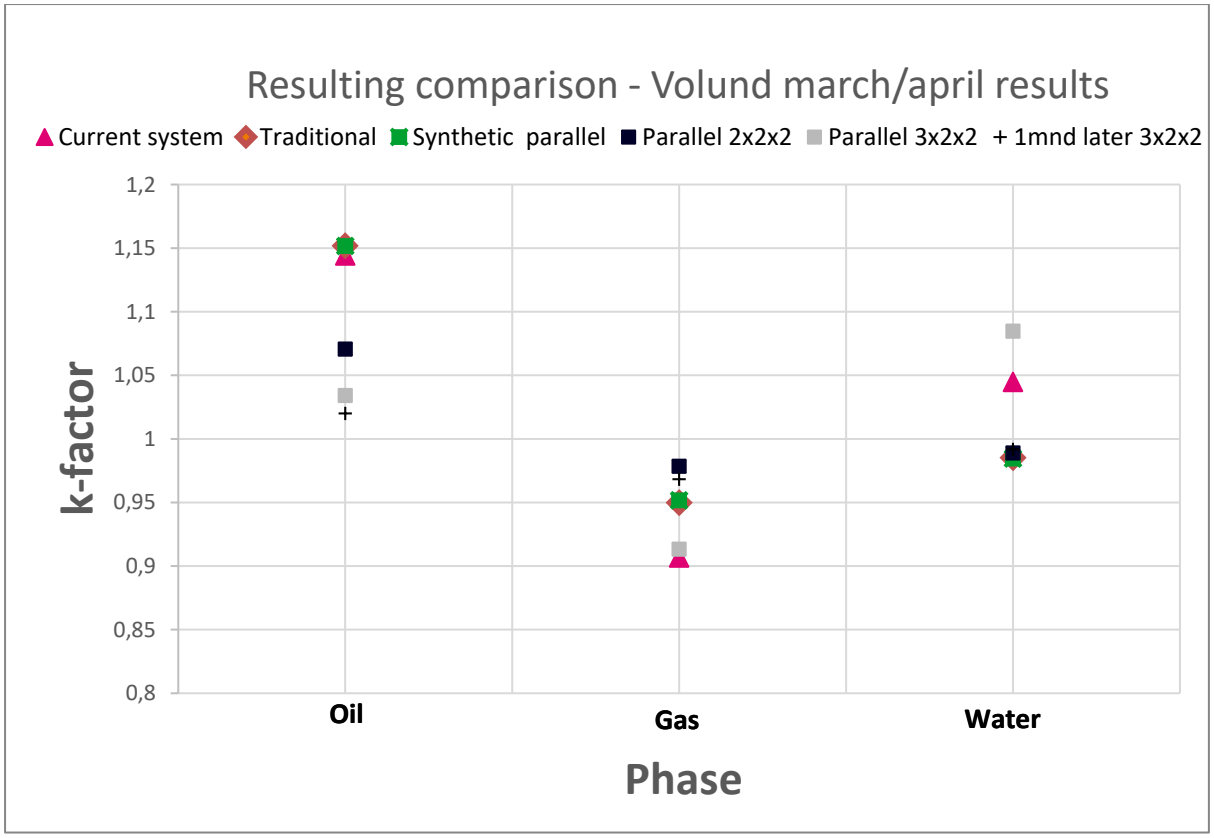


Figure 38 - Result comparison of Volund during March/April calibrations

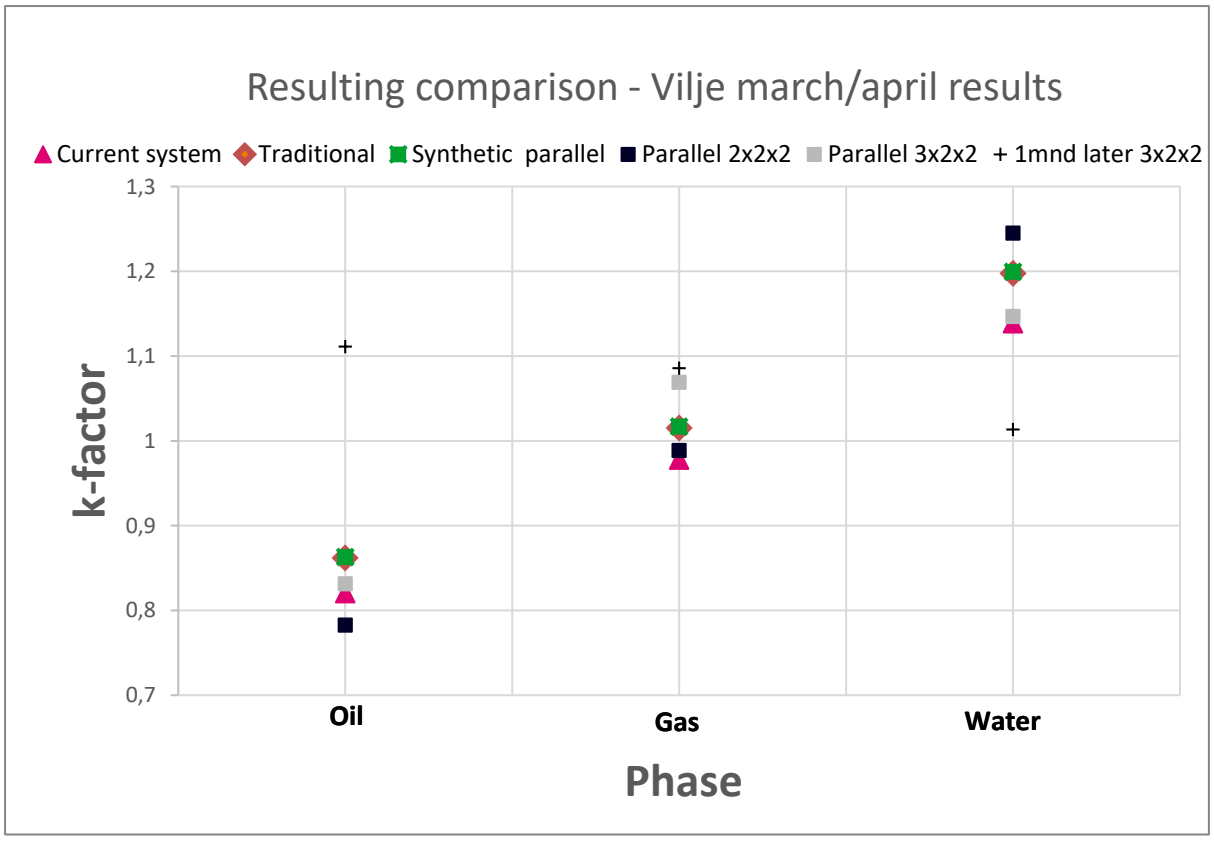


Figure 39 - Result comparison of Vilje during March/April calibrations

7.3 Financial gains of using this algorithm

The primary gains of using the parallel calibration over the traditional method is the reduction in production deferrals which is a consequence of the use of the traditional method.

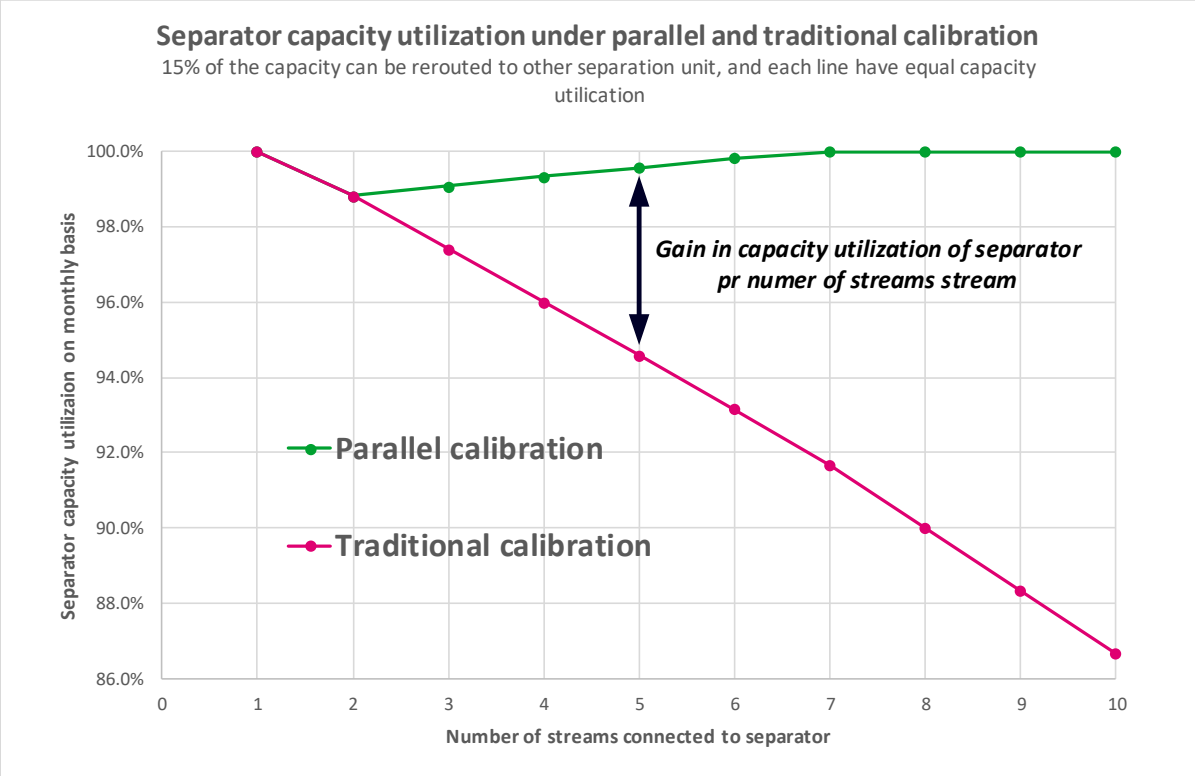


Figure 40 – Qualitative aspects of separator capacity utilization under parallel and traditional calibration

The reduction of capacity utilization of a separator is highly individualized for every single input stream, and there can be significant differences between the streams, but Figure 30 shows a qualitative assessment of the reduction of utilization of the separator, under the assessment that every calibration trial takes 12hours for both traditional and parallel calibration and that 15% of the capacity of the separator can be rerouted to another separation unit and that every month is 30 days. But the main concept that with the use of parallel calibration the more multiphase streams are connected to the separator the lower production deferral will be, which is the opposite of the traditional method where the more lines are connected the higher the production deferrals. In reality the choice of which production lines to reroute and which to perform the calibration on can be chosen in order to not affect the critical producers or if there are some fields with reservoirs sensitive changes to the field, in order to operate the connected licenses in a best possible way. And during the Alvheim late march early April 2019 test the production deferrals during traditional calibration where 9500bbl and deferrals during parallel calibration was 310bbl for the late march early April test, and 6200bbl deferrals vs 50bb in the late April calibration which are a significant reduction in production deferrals[12], which is better than what is expected from the qualitative graph in Figure 40.

[12] Deferral calculations where based on a “Best day performance” use-case developed inside Aker BP ASA

8 Discussion

This chapter will discuss the content of this thesis and give thoughts of what is achieved both in the results created by the algorithm, but more importantly on the development and implementation of the method in the algorithm created, the way forward in improving the method.

8.1 Development

The development of the algorithm is built on layer by layer of abstract function, from the accumulator, which is implemented in stream objects where each stream accumulates each phase. These stream objects are then collected into collection of trials of the same time window. A collection of these trials, which are then used as the basis for a parallel calibration routine, that use the stored and aggregated data in each trial and subsequent stream. And when everything is object oriented all data is readily available for any type of data analysis.

Since I am both a user, customer and developer, the software development cycle closed in on itself and turned into a rapid evolutionary development method. With regards to the unit testing of the data in most of the python scripts created, if the script is run by itself as main, then a unit testing algorithm is executed in order to check both the methods and the algorithm as a whole, to ensure all methods are working as intended, and to simplify both the development and debugging of the software.

By object orientation of classes and algorithm the resulting system has great usability, and transparency both for the execution of the algorithm and further data analysis and development. Many of the methods used in the developed algorithm are solved elegantly and efficiently, but there is always further use of the NumPy library where more efficient changes can be written in the codebase. There is also the opportunity of multithreading when executing larger operations encapsulated in object instances for example during the creation of the digital representation of the streams. Where all trials can be calculated in parallel. but the algorithm performs sufficiently for the execution for the purpose of parallel calibration, The operation of synchronizing data of the different streams and trials is solved very elegantly and uses the efficiency of the NumPy library to find the closest datapoints to calculate against.

8.1.1 Non-available historical datapoints

Datapoints of densities for oil, gas and water is missing for the data repository. But pure net oil in mass rates are available from the metering system. But the mass flow of gas and water on the separator are calculated through a constant density. where the gas stream has a constant standard density. But as it is, the line oil stream density can be inferred from looking at the volumetric oil stream, water cut as well as the pure oil mass stream. But when this codebase is further developed, densities and other rates on the multiphase meters such as calculations of GOR and GVF can be included in the multiphase streams, which are meant to be placed through the flow weighted average algorithm and reported with the resulting values from the algorithm. In order to gauge the k-factor as a function of both the GOR and GVF, but this is to be implemented later, but is of a high relevance for a achieving representative k-factor and provide essential information on the dependencies of the phase fraction as well as important with respect to the uncertainty of the measurement at different phase fractional flows.

8.1.2 Historical logging of each flow computer increments

Instead of logging values and compressing data in PI, each increment calculated by the flow computer should be losslessly stored and made available in the data fusion repository or to be executed on the metering SCADA level, for more precise and better data continuity for calculations. To have a constant time between each datapoint also opens other opportunities and a basis for implementation, more elegant analysis and establishment of a lot better calibration characteristics.

8.1.3 Negligence of physical properties

There is a lot of physical properties neglected in the implementation in order to achieve a workable implementation and to execute and test the parallel calibration methods, this is neglecting the

- water moisture in the gas (the fraction of H2O molecules in the gas) this thesis has not looked into the significance of this.
- Oil in water stream of the separator is not looked into.
- Injected chemicals into the streams between the topside MPFM and Test Separator, since the amount injected during the calculation are neglected during normal calibration.
- The waters streams in both the oil and the water outlet of the separators are in pure volumetric line conditions, and there are no calculations done to standardize the volumes to standard conditions.
- On the separator gas stream a constant standard gas density is implemented, and there should be looked into a calculation of the density of the gas based on either the predicted composition from the PVT model and or calculated through both the velocity of sound from the ultrasonic gas flow meter and the oil density, with regards to the expected standard gas density at the pressure and temperature conditions at each moment.

8.2 Parallel calibration

The parallel calibration method can be used on both single-phase calibration and on multiphase calibration. For any good calibration of flow meters stable conditions is important. If conditions are not stable, then a good calibration is hard to achieve. And when it comes to performing a calibration with the use of the algorithm created in this thesis, is very simple, as shown in Figure 32. The algorithm gives also great flexibility in both choosing the time window, and provide visual diagnostic insight into the data inside the algorithm. One of which is the developmental k-factor. Instead of just accumulating and solving the for one instance at the end of the dataset, the algorithms solves for the k-factors across the trial windows, and through this can provide similar acceptance criteria as is used in the method today. The augmented matrix plotting gives great visual insight into all the data fed into the algorithm, where the conditions as well as the data integrity is shown clearly. And the codebase is ready to have further more intensive properties, when the data is available in the data repository, such as density’s and volumetric phase fractions, and with the flow weighed averaging in the accumulator the data will be representable for the entire calibration and be helpful in establishing how the calibration on each meter is effected by phase fractions.

8.2.1 Trial quality, size and order.

To be able to minimize the length of the trials, but still have significant data to get a representable result, is something to look further into and can further reduce deferred production during a trial, where one of the higher flowing streams are deferred or rerouted.

Look into how smaller variations of flowrates in each trial can be without the matrix going singular. An see if well tests and other normal downtime can be used in the creation and finding potential trial windows.

When creating a trial, for the implementation now it is important to have the combination of streams consistent, so the streams com in the correct order, but an auto-sorter method will be implemented into the algorithm on a later stage.

8.3 Traditional vs Parallel calibration

When looking at the difference between traditional and parallel calibration, the reduction in production deferrals are a significant benefit for the parallel calibration method, but there are other factors as well to take into consideration; Such as that during a parallel calibration a more nominal production is occurring, such that that the calibration is performed closer to normal operation conditions, and less pressure and temperature difference between the multiphase meter and the separator, the flowrates out of the separator are higher which can be beneficial for the uncertainty of the measurement. But with the Parallel calibration there is no longer a one to one relationship of the measurement, which can cause uncertainty and makes the calibration less intuitive and breaks with the established calibration norms that it is based on. But then again there are single phase system of master duty meter calibrations where the master meter are precise meters connected in parallel which work as the reference, this is of course the same here since both the oil and gas streams have two streams each. But on the duty meter side having several on calibration at once is thought provoking.

8.4 The achieved result

Looking at the January result there is a static deviation which can be a product of the constant gas and water density, flashing calculation, or the time window or due to the difference in the raw data point resolution. But when looking at how the algorithm produced both the traditional result and the synthetic parallel, they overlap extraordinarily well which gives great confidence to the method and algorithm.

Looking at the March and April results there was one trial of the traditional calibration trial which had not the best process integrity due to the variation of the oil levels in the separator during the trial, that can be a source of uncertainty this is also shown in Figure 31, but both the start and the end where stopped on approximately the same level. Additionally, there where some non-continuities in the k-factor development that is not investigated to their full extent. But for the synthetic and the traditional result they overlap well for this result as well.

And when it comes to the recalibration in late April the results are as expected, but with a slight change to the calibration on the Vilje stream.

But for both the January and the later test there is a difference of the time windows used between the actual system on Alvheim and the algorithm which can also be a source of the difference, and also the effect of the flashing calculation, due to when a triple stream or dual stream trial is online there is a smaller pressure and temperature difference between the multiphase meter and the separator. But the algorithm performs satisfactorily.

But looking at the results of the created traditional data set and the synthetic parallel dataset which both are based on the same data, the results are satisfying, that the parallel calibration method performs to the same or better than a traditional calibration does, when the trials is a combination of 2 streams, but when even more streams are in one of the trials . And it is the calibration method and algorithm which has been the main point of this thesis, that has the simplified flash implemented. But when a better flashing algorithm is also implemented the lower difference between the multiphase meter and the separator conditions should provide less flashing and the flow meters on the separators are operating closer to their points of higher accuracy and in general the calibration is executed closer to the normal operating conditions, than during normal calibration where there are greater pressure and temperature differences between the multiphase meter and separator, together with lower flowrates through the separator.

8.5 Uncertainty

When it comes to quantifying the extended uncertainty of the method in general, this requires a competence I do not currently have, since this is a linear combination of several accumulated values in a matrix, but if the values accumulated on the single phase system are according to the NPD's regulation, and with regards to these single phase uncertainty, the result of the random uncertainty of the k-factors are calculated as a part of the result basis set for each phase and stream. And assuming the data is normally distributed as both shown in the dual histogram and k-factor development plot and the violin plot the uncertainty is established from this with respect to the measurement systems single-phase uncertainty.

8.5.1 Traditional vs Synthetic parallel calibration

Comparing the uncertainty between traditional and parallel by looking at differences in the January 2019 calibration, the uncertainty between a traditional and parallel calibration executed by the algorithm, shows a very small difference in random uncertainty of the result and the k-factors produced are very similar as shown in table 7.2, this gives confidence in the parallel calibration method. But it is important to root this result on a synthetic combination and underlies the assumption that the systems are linear. Proof that the systems are in fact linear comes from the April results where the traditional result is compared to the 2x2x2 parallel calibration executed, where the best result was on the Bøyla stream show in Figure 37.

8.5.2 Calibrating closer to normal operating conditions

Lower pressure and temperature differences between the multiphase meter and the separator during parallel calibration due to higher use of the capacity of the separator, this lower difference in the process condition lessens the effect of the flashing, and the uncertainty it entails. But more importantly the flowrates through the separator is higher and as such the single phase measurements can be operated with a flowrate better suited to the turndown[13] of the meter.

[13] Turndown / effective range is a dimensioning figure of the operational area of a flow meter. And is the fraction of the maximum flow divided by the minimum flow of the meter can measure within specified uncertainty

8.5.3 Numerical uncertainty in the parallel calibration method

From the result of the comparison of the traditional and the synthetic calibration the resulting values, both due to the discreet numerical integration of varying time delta, as well as the method takes in combinations of several streams at the same time.

8.5.4 In calculation of flashing

Until a good flashing algorithm is implemented into the data creation of the multiphase streams to do a proper flash of hydrocarbon components within the stream, there will be a systematic error of the precision and offset the k-factors calculated, but by implementing a well working either through commercially available application or as an algorithm implemented within the stream through a petrochemical model or a table lookup of simulated data similar, and to also have flashing algorithm which is unique for each stream, since each stream have its own composition based, and by having the measured or inferred density's from both the multiphase and the single phase measurement correcting each other can provide additional features for this type of data analysis.

8.5.5 Accumulation quality

The raw datapoints of mass flow rates accumulated has a varying time between each datapoint and having a sufficient density of datapoint for the is worth documenting, this can be done by plotting a histogram of the time difference between each datapoint accumulator, and due to data compression algorithm on the OSI soft PI system if the values remain stable over longer periods a new datapoint may not be available for a longer time period. But by the use of the augmented matrix plotter a histogram of the time difference between each datapoint can be documented and plotted with a logarithmic y scale to visualize the spread for the time differences between the data points.

8.6 Further work and development

There is also of more work which can be used with respect to the post calculation of behavior multiphase flow streams, but this is just the beginning of using data science and to use the data availability which are available now through repositories such as Data Fusion by Cognite, this thesis is simply the tip of the iceberg.

8.6.1 Automatic multi trial combination

There are $T!$ (Factorial) number of combinations, and the parallel calibration method can be done with all these different trial combinations. So for the 3 stream calibration unique trial combinations can be executed and matched towards each other on order to increase the quality.

8.6.2 Multi-dimensional multiphase meter calibration characteristic

To give each phase just a scalar factor for the calibration can be a crude method of calibrating complex instruments there could be done a lot of work looking into creating a multi-dimensional calibration characteristic for the multiphase meters, where there are more dependent variables. It is also experienced that the uncertainty of the multiphase meters are dependent on the velocities, flow scheme phase fractions in the streams, as covered in earlier

chapters. And to take this into account and give a deeper understanding of the meters can additionally reduce the uncertainty with regards to the meters in question.

8.6.3 Increment database in flow computer to CDP

When it comes to the database which stores the increments in each accumulation all the increments for each second would be great if this was opened up to CDP, and with a datapoint for each second would open up opportunities to do more in-depth calibration as well as being able to verify and parallel calculate rates and do better data analysis with a constant iteration time.

8.6.4 Sliding time-window approach

The result of the calibration with a sliding window approach would be interesting to see the result, this will also give more consideration to the flow weighted average of the different fractions at any time during the sliding time window.

8.6.5 Visualizing the k-factor over runs

To be able to create the k-factor development over the run time, there are a couple approaches to this, the method can scan through all runs at the same time, giving the plots such as the result of the synthetic and their by mimic the tradition calibration, or all but one trial set can be held constant on a value acceptable accumulated value and vary one trial data. Or there can be performed a multi-dimensional calibration creating an $T \times S$ dimensional matrix of k-factors which can be visualized in selected ways. But for this thesis the main method used is the simultaneous scan over every trial at the same time to mimic the tradition result, to have a comparable reference. But there are some further developments that can potentially provide some deeper insight.

8.6.6 Soft-sensor of multiphase meters

The use of this algorithm and methodology can be used to create an estimator of the k-factor through a moving trial window with respect to previous trial. And thus, have a constant calculation of k-factors, and if this where to change a new parallel calibration rerouting procedure could be executed and give new k-factors.

8.6.7 Calculate separator streams to multiphase conditions

In the implemented method, the multiphase meters are flashed to the separator streams. But to look into how the single phase separator streams can be back calculated into multiphase meter conditions should provide additional insight, the implementation now has the comparison done at separator conditions, and it is the multiphase meters which are of interest in the calibration.

8.6.8 Petrochemical calculations program development

There are many commercially available programs and modeling tools available, but in the spirit of data liberation, open source libraries with regards to flow calculations and flashing and PVT algorithms is of great interest both for applications and for educational and research purposes, which this algorithm can benefit from.

8.6.8.1 Flashing and PVT calculation library for python

Develop Flashing and PVT calculation library for python, to be run inside the multiphase streams with given composition and pressure and temp deltas between stream and separator. In order to perform a flash to separator conditions.

Or more advances a compositional calculator based on well test and expected decline curve from GOR to be flashed to multiphase meter conditions, and further flashed to separator condition, further corrected against the measured oil density and the calculated gas density where the differences is taken into account to correct the flashing algorithm. The gas density has a possible inference method based on the velocity of sound from the ultrasonic flow meter.

8.6.8.2 Soft-sensor for gas density based on VOS, Temperature pressure and assumed composition

Create an open source soft sensor for density estimation for gas flowrates based on velocity of sound. Velocity of sound is easily gotten through ultrasonic flow meters, which are frequently used on hydrocarbon gas streams in both wet- and dry-gas application. These ultrasonic flow meters in addition to the fluid velocity and profile. Also give out the velocity of sound through the gas media, and the velocity of sound together with temperature and pressure and an initial guess of the composition can be used to infer the composition and through the composition the standard density of the gas.

8.6.9 Digital twin giving further insight than planned

When development of the algorithm, the approach of creating a numerical digital representation of the streams in the calibration system has revealed further insight and opportunities than just the parallel calibration. When the code is structured in a physical object-oriented way, it enabled newer insight into checking the state of the k-factors by using the same trial vectors in another way to ensure that the mass balance of the system is intact and their by alternative insight into. And this gives then a “free” soft sensor of the health of the k-factors for the MPFMs in question.

Developing the software and calculations used inside the digital twin is also a very good method of familiarizing yourself with the system and the data you are working with, development of this kind of software for training purposes should provide benefits for engineers and technicians working with the system to gain a more in-depth understanding as well as opening up for more creative problem-solving and data analysis.

8.6.9.1 Calibration by difference

During the development of the parallel calibration algorithm a new multiphase meter on the Bøyla flow line was installed, it was change from a separate brand to a MPM Multi phase meter, during the initial flows on the meter there where processes issues that prevented a normal calibration to be performed, but the k factors for the other streams (Vilje and Volund stream) was established and stable and with the use of the single trial calculations of the streams the apparent k-factor for the new Bøyla MPFM was established with little to no change in the code. By rearranging the equation from (6.2) to the equation (8.1).

$$k_{p,Bøyla} = \frac{\left(m_{p,ref} - \left[m_{p,Vilje} \cdot k_{p,Vilje} + m_{p,Volund} \cdot k_{p,Volund}\right]\right)}{m_{p,Bøyla}} \tag{8.1}$$

8.7 Proposal for new structure for parallel calibration

Although this algorithm has been tested with 3 multiphase inlet streams the mathematics are the same with more that 3 streams, this will also entail more trial combinations, but all information indicate that it is a possible solution for the future, if a production hub is connected to many fields. But to have a smaller separator for a one to one relationship is also an opportunity, but to use bigger main separators for proper separation of the liquid phases can be beneficial. But a proposal is shown in Figure 41 if the parallel calibration method is used.

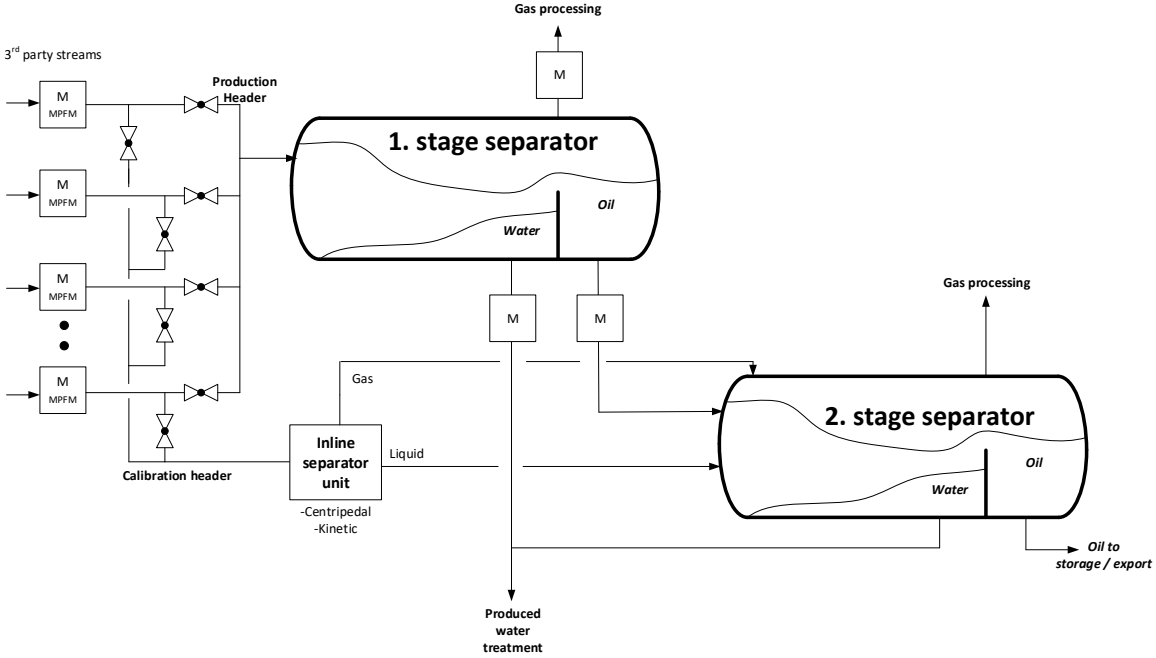


Figure 41 - New structure for parallel calibration concept

9 Conclusion

The use of data science on industrial data in the manner done in this thesis shows the great opportunities which digitalization can achieve through liberation and contextualization of data. The algorithm developed here can be of great benefit when it comes to the use of existing processing and transport infrastructure through the third-party access focus in the maturing parts of the Norwegian continental shelf. I feel this is just the tip of the iceberg of additional benefits and insight through data science on historical and real-time data streaming from the asset intensive industries.

This thesis provides details about the purpose and technical aspects of sensors and equipment, used in both single and multiphase allocation measurements of oil and gas. An algorithm solving a parallel calibration of multiphase meters has been developed, with implemented quality assurance aspects of the data used and calculated, addressing both the process conditions, data integrity and the random uncertainty of the result. The calibration uses real data from Aker BP-operated Alvheim and executes both actual and synthetic calibrations. The result of several calibrations is discussed in detail. Including the methods and their compared results and uncertainties. The assumptions of the method concerning the mass balance of the system, and the proportionality between single and multiphase, seem to hold its ground empirically. When it comes to statistical basis and assumption that stabilized values are normally distributed, this seems also to hold for the calibration where a stable result is achieved. Future work and development of the method and algorithm are also discussed in detail.

The effects of using this algorithm compared to the traditional provide a significant reduction in production deferrals, to the point where the deferrals are almost non-noticeable from normal production, which has a significant economic effect. And the result from the calibrations performed in the start and end of April 2019 had a reduction of above 15000bbl of oil. With a assumed realized oil price of 70 USD/bbl[14] and an exchange rate of 8.7 USD/NOK[15] equates to above 9 million Norwegian Krones over the course of one month. An additional benefit is that the code developed during this thesis provide the opportunity to analyze the health of the multiphase meters, and gives grounds for a more condition based monitoring of the multiphase meters and potentially increasing the time between calibrations, or by increasing the accuracy by addressing the need for a calibration at an earlier stage. With the future plans on Alvheim to potentially tie back more fields, then this calibration method has an even bigger potential in reduction in production deferrals. The algorithm is implemented with this in focus and solves a general case, with as many streams as the user can physically have installed, meaning the algorithm is scalable. The code is also highly portable and can run on a metering SCADA level with a minimal amount of extra code during implementation.

A new algorithm has been created in this thesis with a new way of looking at utilization of stored and streaming industrial data. And I'm quite proud of what has been achieved in this thesis, in the last 4 months. And with a fulltime job at the same time as I've created the algorithm and written this thesis shows the power of digitalization.

[14] In April 2019 the Brent Oil Spot price was around 70 USD/bbl.

[15] the exchange rate of US dollar to Norwegian kroner 12/05-2019

References

[1] A. B. ASA, "Aker BP ASA," Aker BP ASA, [Online]. Available: https://www.akerbp.com/. [Accessed 11 05 2019].

[2] A. ASA, "Aker ASA," Aker ASA, [Online]. Available: https://www.akerasa.com/. [Accessed 11 05 2019].

[3] C. AS, "Cognite," [Online]. Available: https://cognite.com/. [Accessed 11 05 2019].

[4] A. B. ASA, "Alvheim," [Online]. Available: https://www.akerbp.com/produksjon/alvheim/. [Accessed 11 05 2019].

[5] M. o. P. a. E. a. N. P. Directorate, "EXPLORATION POLICY - NEW DISCOVERIES – EFFECTIVE USE OF INFRASTRUCTURE," 29 04 2019. [Online]. Available: https://www.norskpetroleum.no/en/exploration/exploration-policy/#effective-use-of-infrastructure.

[6] P. S. Foundation, "Python Software Foundation," Python Software Foundation, [Online]. Available: https://www.python.org/. [Accessed 11 05 2019].

[7] M. o. p. a. E. a. N. P. Directorate, "norways petroleum history," 29 04 2019. [Online]. Available: https://www.norskpetroleum.no/en/framework/norways-petroleum-history.

[8] N. government, "Norway’s oil history in 5 minutes," 29 04 2019. [Online]. Available: https://www.regjeringen.no/en/topics/energy/oil-and-gas/norways-oil-history-in-5-minutes/id440538/.

[9] T. N. P. Directorate, "About us," 19 04 2019. [Online]. Available: https://www.npd.no/en/About-us/.

[10] M. R. B. CURTIS H. WHITSON, PHASE BEHAVIOR, Richardson, Texas: Society of Petroleum Engineers Inc and U. Trondheim, NTH., 2000.

[11] N. P. Directorate, "The measurment regulation / Måleforskriften - Regulations relating to measurment of petroleum for fiscal purposes and for calculation of CO2-tax," 2001.

[12] M. o. P. a. Energy, "Regulations relating to the use of facilities use by others," 2016. [Online]. Available: https://www.npd.no/en/regulations/regulations/facilities-use-by-others/. [Accessed 03 05 2019].

[13] S. H. Gustasven, "Expansion of test facility for flow measurement on drilling fluid - Appendix AA - Basic concepts of fluid dynamics and rheology," Telemark University College; Faculty of Technology, Porsgrunn, 2015.

[14] S. Corneliussen, , J.-P. Couput, E. Dahl, E. Dykesteen, K.-E. Frøysa, E. Malde, H. Moestue, P. O. Moksnes, L. Scheers and H. Tunheim, HANDBOOK OF MULTIPHASE FLOW METERING - ISBN 82-91341-89-3, Oslo: The Norwegian

Society for Oil and Gas Measurement, The Norwegian Society of Chartered Technical and Scientific Professionals, 2005.

[15] TechnipFMC, "MPM – White Paper 00 - How The MPM Meter Works," TechnipFMC, 2017.

[16] E. Dahl, E. Dykesteen, E. Jacobsen, E. Malde, F. Flåten, H. Tunheim, M. Brandt, R. Albrechtsen, O. Albrechtsen, O. Vikingstad, S. E. Corneliussen and T. Hjorteland, Handbook of Water Fraction Metering, Oslo: Norwegian Society for Oil and Gas Measurement, 2004.

[17] Ø. L. B. Arnstein Wee, "In-Situ Measurement of Fluid Properties and Integrity Verification for Multiphase and Wet Gas Metering Applications," North Sea Flow Measurement Workshop, 2010.

[18] N. s. f. o. a. g. measurment, "Handbooks and uncertainty programs," [Online]. Available: https://nfogm.no/handbooks-and-uncertainty-programs/. [Accessed 29 04 2019].

[19] N. i. o. S. a. technology, "SI redefinition - Kilogram: The Kibble Balance," [Online]. Available: https://www.nist.gov/si-redefinition/kilogram-kibble-balance. [Accessed 30 04 2019].

[20] Justervesenet, "Det nye SI-systemet trer i kraft 20. mai 2019," [Online]. Available: https://www.justervesenet.no/det-nye-si-systemet-trer-i-kraft-pa-verdens-metrologidag-20-mai-2019/. [Accessed 19 04 2019].

Appendices

Appendix A < Task Description, WBS and GANTT>

Appendix B <Non-public documents>

Appendix C <Digital Representation of fluid streams>

Appendix D <Parallel calibrations algorithm>

Appendix E <Calibration execution and result January>

Appendix F <Calibration execution and result April>

1 Appendix A – Task Description, WBS and GANTT

Contents

1 Appendix A – Task Description, WBS and GANTT .....
2 Thesis Task .....
3 WBS.....
4 GANTT.....

2 Thesis Task

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Parallel calibration of multi-phase flow meters (MPFM) based on measurements of phase streams in separators

USN supervisor: Saba Mylvaganam

External partner: AkerBP (Torbjørn Selanger)

Task background:

On the Alvheim FPSO operated by Aker BP, there are multiple fields connected to the Alvheim FPSO for processing and export. For 3 of these field (Volun, Vilje and Bøyla) the ownership allocation of the oil and gas is established by a production measured on a topside MPFM for each respective field. Measured values of separated phase streams out of a common separator (3rd party separator) are used as reference to calibrate the MPFMs.

Under calibration operations for the MPFM, only the flowline being calibrated can flow through the separator, and the other two flowlines have to be rerouted to another separator (Alvheim separator) with limited capacity. To avoid exceeding the capacity of the Alvheim separator, the production has to be reduced resulting in deferrals. Calibration takes place every month and lasts for approximately 12 hours for each flowline. This Thesis will look into verification or calibration multiple MPFMs at the same time to limit production deferrals.

Task description:

Main tasks:

- Give a detailed overview and purpose for the ownership allocation system, as well as the sensors and equipment used.
- Develop and explain an algorithm which tries to solve the apparent calibration method by running three runs of two fields through the separator at the time.
- Execute the calculation algorithm with actual or synthetic dataset.
- Discuss result of the method
- Compare method and uncertainties between this method and the traditional calibration method
- Assessing the calibration/verification method and list up future work to be looked into.

Submitting a Master Thesis following the guidelines of USN with necessary programs and including a well-documented and complete set of all experimental data from the measurements

Student category: (EET, EPE, IIA or PT students)

IIA – Master students

Practical arrangements:

The masters thesis is open for the public. The sensitive information regarding production, uncertainties and cost can be sensitive data and will be in appendices which is not open to the public, or considered with Aker BP ASA, in accordance to Aker BP ASA *Specification for Securing Aker BP Data Do, c no.: 78-04-000135.*

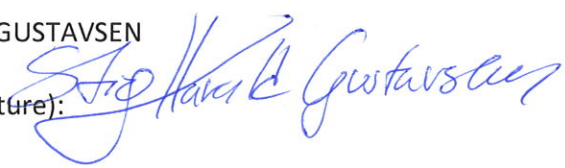
Supervision:

The student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

Signatures:

Supervisor (date and signature):

Student : STIG HARALD GUSTAVSEN

Student (date and signature): 31/01-2019 

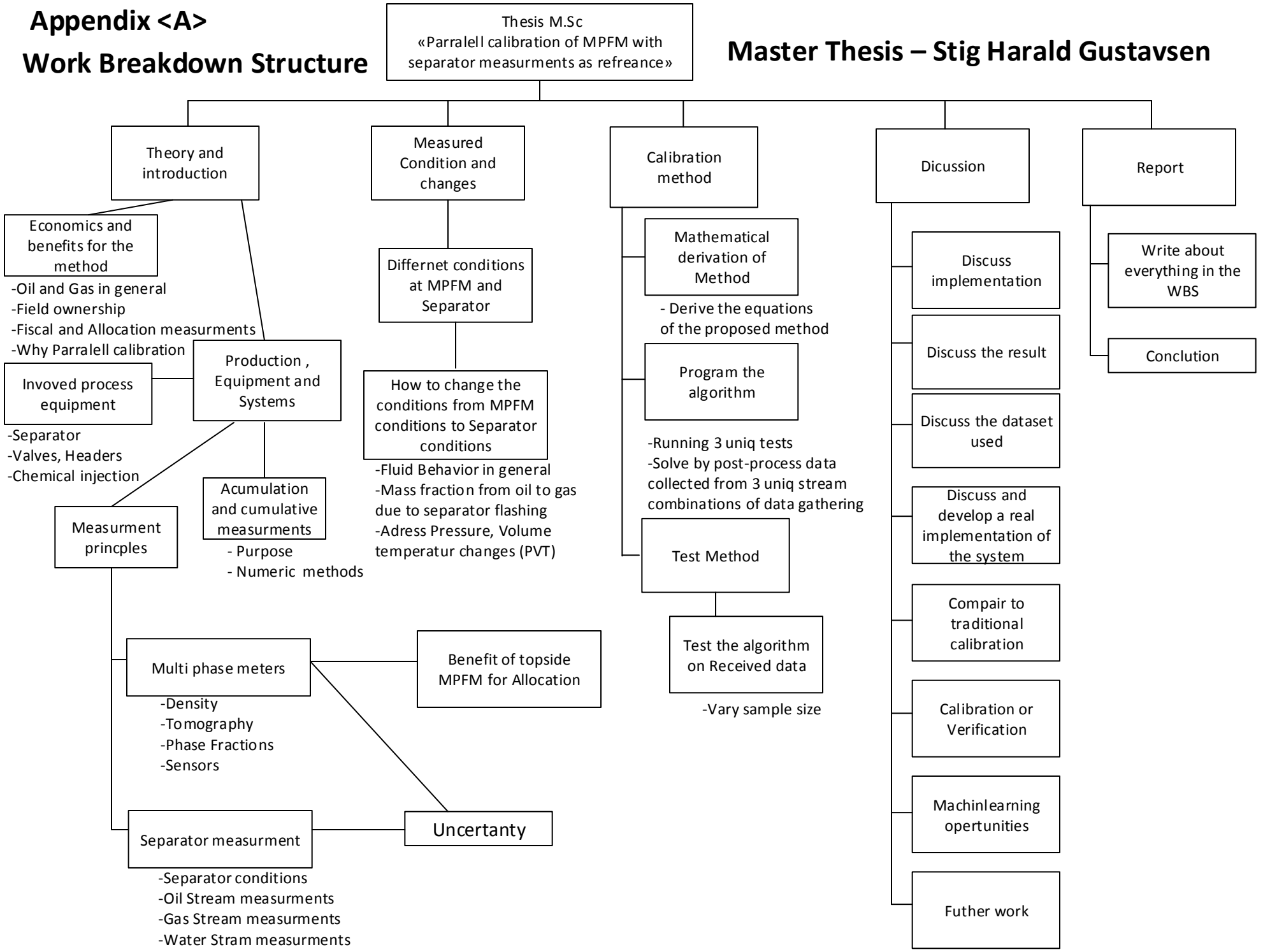
Address: Kjølnes ring 56, NO-3918 Porsgrunn, Norway. **Phone:** 35 57 50 00. **Fax:** 35 55 75 47.

3 WBS

Appendix <A>

Work Breakdown Structure

Master Thesis – Stig Harald Gustavsen



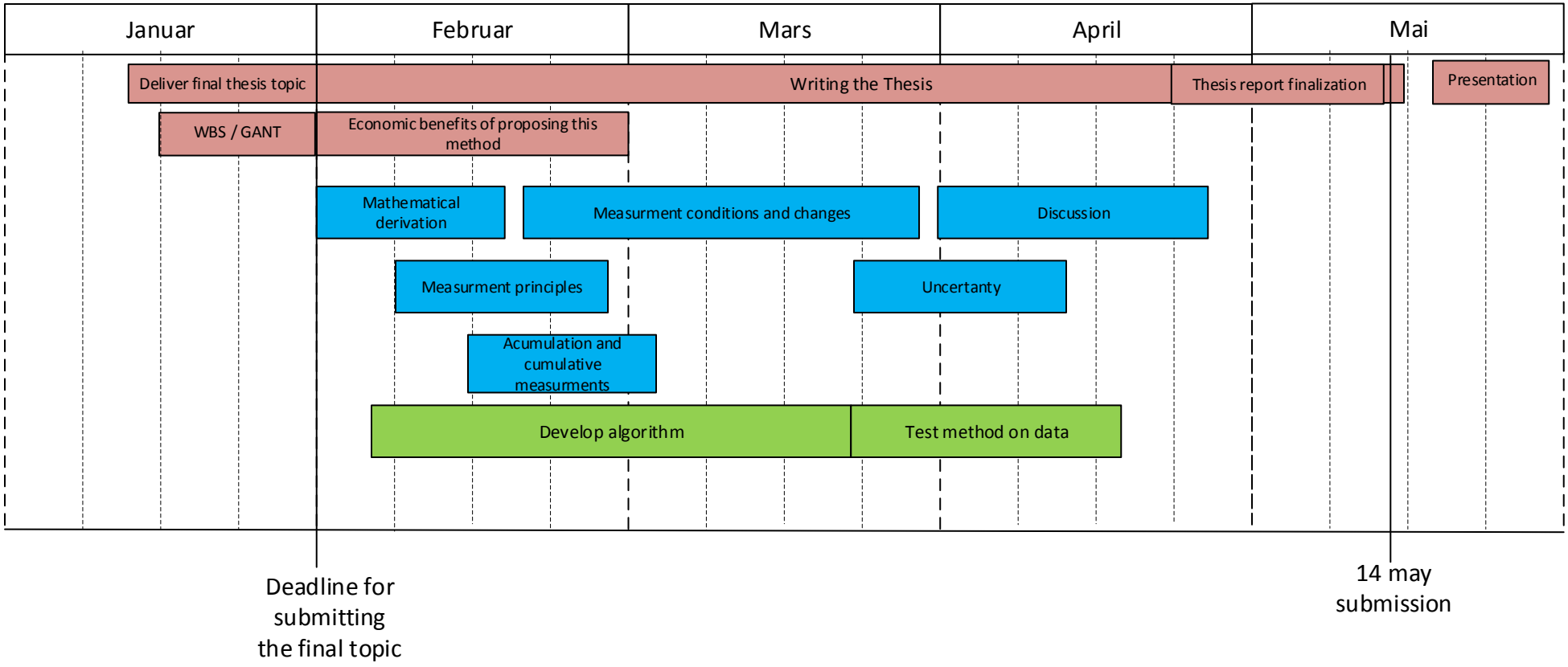
4 GANTT

Master Thesis

Stig Harald Gustavsen

Appendix <B>

Master Thesis 2018
«Parralell calibration of MPFM with separator measurments as refreance»



1 Appendix C – Digital Representation of fluid streams

Contents

The content of this appendix will go into the technical details about the software development of this thesis and go into detail implementations of the digital twins of streams and the calculations used to create these digital representations of the state's important and to prepare and prepare and have all data needed for further processing in the calibration algorithm created in this thesis. But first the raw data from the sensors through the Cognite Data Fusion repository needs to be processed and prepared for further use. First it will describe the accumulation which occurs on every single-phase stream rate and the go into the details concerning the multi-phase and separator streams, and the calculations performed inside these instantiated objects.

1 Appendix C – Digital Representation of fluid streams 1

2 Software development 3

2.1 Python framework 3

2.1.1 Code and calculation execution efficiency 3

2.2 Cognite Data Fusion 3

2.2.1 Datapoints, timestamps and time series data 4

2.3 Applied calculations 4

2.3.1 Calculating towards the closes datapoint 4

2.4 Unit testing 4

3 Accumulator 5

3.1.1 The flow weighted average (FWA) 6

4 Streams 7

4.1 Multi-phase streams 7

4.2 Separators Streams 8

4.3 Trial creation and object orientation 10

4.4 Datapoints toolbox 12

4.5 Trial time window locator 12

5 Source code – Accumulator 14

6 Source code – Streams 18

7 Source code – Datapoint toolbox 22

8 Source code – Time Window locator 25

2 Software development

In general, the development of the software methodology of the software in this thesis has been in a evolutionary method. And has started from the lowest functional abstraction which in this case is the accumulator. The general-purpose accumulator was created, and then this was used as a crucial building block of the stream's classes. And then the streams abstractions were used as a comprehensive and object-oriented data basis for the data analysis and calculations carried out in the parallel calibration algorithm. This Appendix will go into details.

2.1 Python framework

The entire codebase developed for this thesis in the python language, with sensor / timeseries datapoints from the Cognite python SDK[1] Data manipulations and calculations where done with Pandas[2], NumPy and the non-linear solver also used Scipy[3].

2.1.1 Code and calculation execution efficiency

Pure Python is a dynamically typed programming language, where the entire execution framework is working on a high abstraction layer, which makes it easy to type compared to the compiled language, but being a dynamically typed language comes with a cost in execution time and the computational resource usage, where the memory allocation is distributed over the entire working memory of the computer, this causes higher strain on the computer when looping through data to when it comes to executing calculations. There are frameworks used which tries to reduce the resource needed for calculations, by taking the calculation down to a lower computational level, this is where the python library NumPy comes in. The calculation core of the NumPy library is written in C; with efficient memory allocation and execution reducing the calculation time of, C is a compiled language, with the calculation execution running close to the hardware with efficient memory allocation, and the ability to vectorize the data into NumPy vectors and or matrices, and using techniques for linear algebra to execute code to a much higher degree of efficiency than the dynamically typed language Python. The core of NumPy is also implemented into Pandas library used to develop the algorithms, by simplifying data manipulations and present the resulting values for further processing or display. But back to computational efficiency, there is though cases where the loops and algorithms cannot be vectorizes where the efficiency of the algorithms is crucial and a potential for the use of Numba and JIT[4].

2.2 Cognite Data Fusion

Aker BP ASA uses a data repository from Cognite AS called Cognite Data Fusion, for a lot of their digitalization projects, and the CDF is a data repository containing the real-time and

---

[1] Source code for Cognite Python SDK at: https://github.com/cognitedata/cognite-sdk-python

[2] More information about the data analysis tool pandas on https://pandas.pydata.org/

[3] Information about NumPy and SciPy can be found in https://docs.scipy.org/doc/

[4] Numba with the JIT compiler which can aid in reduction of execution time. And it is also great to be able to execute the algorithm in real-time, but Numba and JIT has not been used but still mentioned due to possible future use. http://numba.pydata.org/

historical datapoints from the majority of the sensors and aggregate data used to develop the both the traditional method as well as the parallel calibration method. The data form CDF is received through their Python SDK, which is open source and available on the package manager for Python packages PIP, and the source code for the SDK is available on a Public GitHub.

2.2.1 Datapoints, timestamps and time series data

The application uses the Cognite Data Fusion as a single source for the timeseries data points used for all the calculations. Data Fusion gets the timeseries datapoints form the OsiSoft Process Information (PI) server, and PI has compression algorithm activated on the tags used in this thesis, which only stores new datapoints if it changes more than a certain amount, which intern causes the time between the datapoints received to vary.

The timeseries data can be received form the Data Fusion repository in both as raw datapoints or as an aggregated for of datapoint such as the average, min or max value during a time granularity. But the raw datapoints can vary with the timestamps. And in AkerBP's case it is the PI by OSIsoft system that work as the repository that Data Fusion collects it timeseries data from. But if an aggregate function which are simpler to work with, where timestamps are indexed toward each other through a granularity property for simpler operations on the data. But the data operations in this thesis will not be working with the aggregate time series, but on the raw datapoints from the Data Fusion where each value comes with a unique timestamp for that specific value, and other operations has to be done to find the closest other datapoint in order to perform a real-time calculation, but all calculations where done on "real" raw datapoints with a varying time between each sample.

2.3 Applied calculations

When it comes to applying calculations where the timestamps are not necessarily the same, and the operations of synchronizing timestamps of the data. Calculations on the Oil stream and water in oil stream are calculated on the same timescale, and the water outlet is calculated on their own term and united on the aggregate accumulated level, this is to simplify the level of the data, because it is a more elegant solution to synchronize the aggregated data, and not implement the calculation on the accumulators increment level.

2.3.1 Calculating towards the closes datapoint

The way it finds the closes data point to a given timestamp is by detecting slope change in the following function where there are two datapoint series the main with *n* datapoints and timestamps the other with *m* timestamps.

2.4 Unit testing

When it comes to testing and unit-testing of the software the main functions as a standard has implemented a simple testing procedure at the end of the related code segment, which is executed if the only the related code is executed as main.

3 Accumulator

The main function of all the different phase streams is the individual accumulation of flow rates for each given phase, in both multi-phase streams and in separator streams. The principle of accumulation of flow and why is described in the main thesis but in this appendix will dive into the actual implementation and code where this is performed. The accumulator has been structured and object oriented in order to perform the same code on each of the phases, so if the integration method used in the algorithm has to change there is only one place this needs to change. But this core feature of the digital representation of the streams within a time window is also executed in python through Pandas and NumPy libraries, and all the values for each of the timesteps between each datapoint, and each increment are all stored an available for any purpose. But in general the accumulator takes in a pandas data frame of timestamps and raw datapoint values form the Cognite data fusion repository and first creates a row of time deltas between each of the timestamps, the timestamps for each the datapoints are in a Unix timestamp format in milliseconds resolution, so to get the time deltas (dt / Δt_n) in second, the time differences are multiplied by 1000, in order to get the values in seconds. The raw values in to the accumulator needs to be in rates pr hour, and the accumulator then creats an pr second rate (PrSecRate / $\dot{m}_n$) where the raw flow rate value in pr hour is divided by a 3600. This pr second rate ($\dot{m}_n$) and time delta (dt /Δt_n) is coped to a normal NumPy array and forms the data used for in the numerical integration.

$$m_n = \sum_{n=1}^{T} [increment_n] = \sum_{n=1}^{T} \left[\frac{\dot{m}_{n-1} + \dot{m}_n}{2} \cdot \Delta t_n \right] = \sum_{n=1}^{T} \left[\frac{\dot{m}_{n-1} + \dot{m}_n}{2} \cdot \{t_n - t_{n-1}\} \right] \quad (3.1)$$

The numerical integration implemented in the accumulator class is the trapezoidal method, shown in equation (1.1), the reason why use the trapezoidal method is that through a simple trick removes numerical voids occurring between each datapoint which is by taking the mean of two datapoints and using this as mean value as a trapezoid between the data points, when there is assumed that there is an analog decline or increase between each value, as qualitatively described in Figure 1.

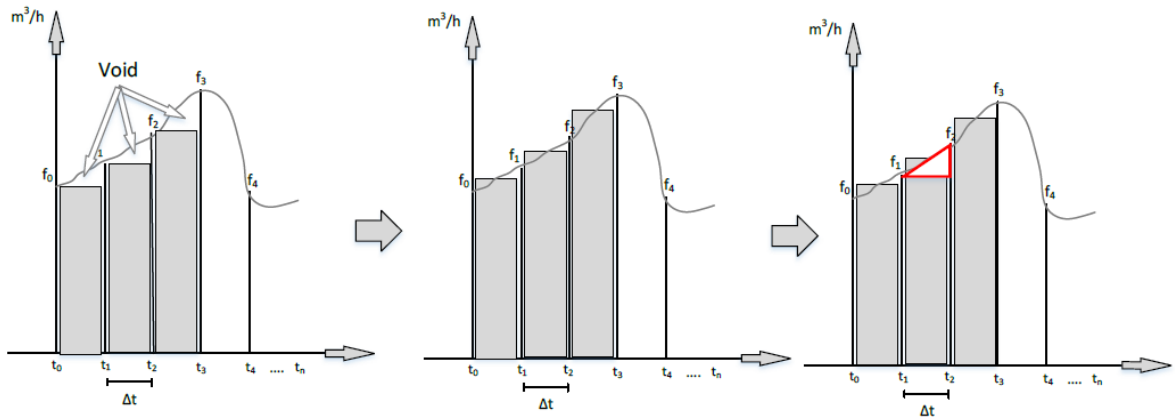


Figure 1 – Qualitative description of increments between midpoint and trapezoidal numerical integration method

All values calculated by the numerical integration method together with increments, time deltas, and the elapsed time of the accumulation, which is the cumulative sum of the time deltas are then collected into a the same data frame and stored to the instantiated object,

Figure 2 shows a portion of such a data frame . It is also worth noticing that later in the more abstract stream classes some accumulators are added together, and this operation is synchronized through the elapsed time of the accumulator and is also used in the creation of a synthetic data set for comparison between traditional and parallel calibration.

```
In [26]: Calibration.TrialStreams[0][0].OilMass.data
```

Out[26]:

	timestamp	value	dt	PrSecRate	incr	cumulative	datetime	elapsed
0	1554002930531	141420.732117	12.043	39.283537	473.091632	0.000000	2019-03-31 03:28:50.531	12.043
1	1554002939563	32649.573441	9.032	9.069326	218.361528	218.361528	2019-03-31 03:28:59.563	21.075
2	1554002948596	19412.231216	9.033	5.392286	65.315872	283.677400	2019-03-31 03:29:08.596	30.108
3	1554002960639	14842.492981	12.043	4.122915	57.295784	340.973184	2019-03-31 03:29:20.639	42.151
4	1554002969671	6441.761456	9.032	1.789378	26.699915	367.673099	2019-03-31 03:29:29.671	51.183
5	1554002978704	2654.098129	9.033	0.737249	11.411514	379.084613	2019-03-31 03:29:38.704	60.216
6	1554002990747	1110.552638	12.043	0.308487	6.296901	385.381514	2019-03-31 03:29:50.747	72.259
7	1554002999779	897.749987	9.032	0.249375	2.519304	387.900818	2019-03-31 03:29:59.779	81.291
8	1554003008812	3199.685383	9.033	0.888801	5.140574	393.041392	2019-03-31 03:30:08.812	90.324
9	1554003020855	4432.862206	12.043	1.231351	12.766496	405.807888	2019-03-31 03:30:20.855	102.367
10	1554003029887	3030.760722	9.032	0.841878	9.362700	415.170588	2019-03-31 03:30:29.887	111.399
11	1554003038920	32203.473511	9.033	8.945409	44.204283	459.374871	2019-03-31 03:30:38.920	120.432
12	1554003050963	83388.069305	12.043	23.163353	193.342910	652.717781	2019-03-31 03:30:50.963	132.475

Figure 2 – The head of Pandas Data frame of an implemented Accumulator object.

3.1.1 The flow weighted average (FWA)

The accumulator class also have implemented a flow weighted average algorithm for creating flow weighted average representations of the intensive variables for the system. This is used when there are intensive variable or any other property that change with the process conditions, and in order to have a representative value of such a variable for a batch within the accumulator the value has to be weighted and averaged across the timespan of the measurement. Which is done by the formulas described beneath, where (3.2) is the analytical case, and (3.3) is in the discrete form and implementable. But in essence the flow weighting is done through multiplying the variable of interest $x_i(t)$ is the index i of the value in question such as the fraction of a component, and t is the time at which the measurement is done and the other figures are the mass flowrates, and thus it is the mass-flow weighted average calculated for the specific variablex_i during the time window T.

$$FWA = \bar{x}_i = \frac{\int_T \left\{\frac{dm(t)}{dt} x_i(t) dt\right\}}{\int_T \left\{\frac{dm(t)}{dt} dt\right\}} \tag{3.2}$$

$$FWA = \bar{x}_n^i = \frac{\sum_{n=o}^{T} \{\dot{m}_n x_n^i \Delta t_n\}}{\sum_{n=o}^{T} \{\dot{m}_n \Delta t_n\}} \tag{3.3}$$

4 Streams

When it comes to the fitting of the raw data to a data set the correct tags and calculations are needed which can be unique for each implementation of the algorithm. But for the time being on the Alvheim field, from the information which is available in the Cognite data fusion repository, and with more information and better flashing algorithms the only place needed to do any changes to the flashing algorithm or performing any other calculations are done within these derived streams classes.

4.1 Multi-phase streams

By having an abstract understanding of the functions of the different equipment involved in the execution of multiphase meter calibration an important abstraction is to see where physical functions are used more than once. And a multiphase meter is something which can occur many times in the system in question.

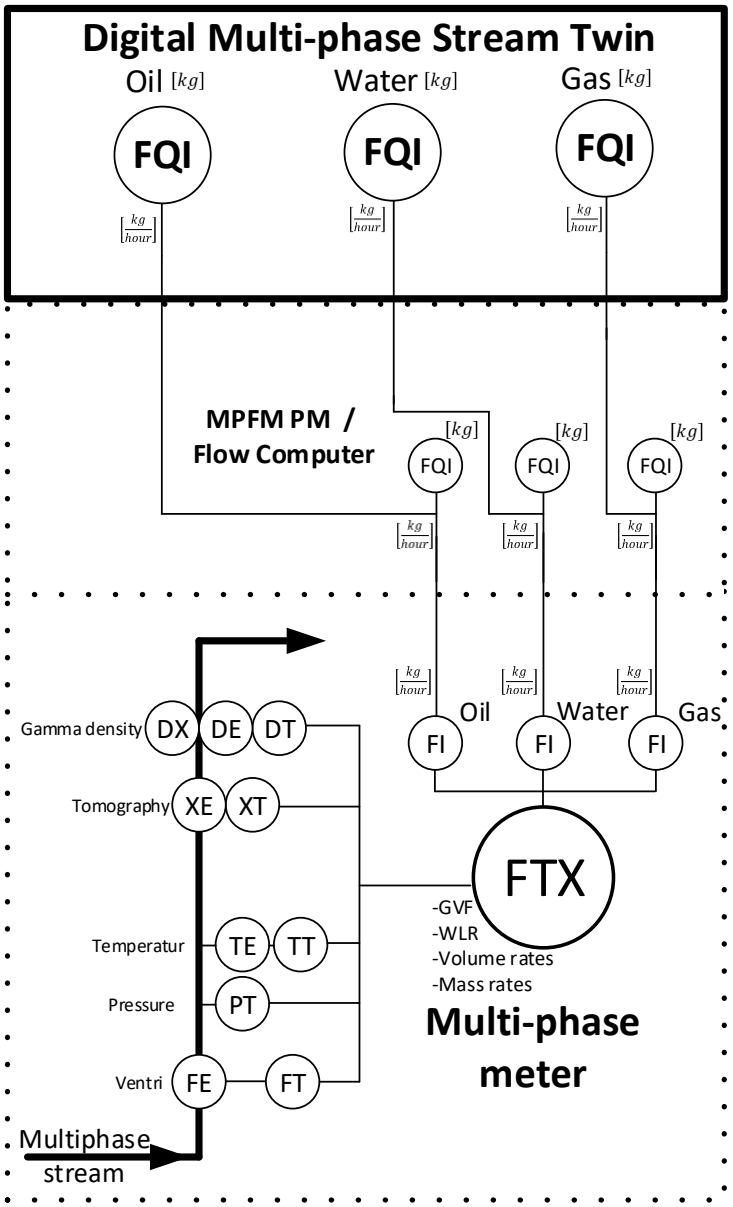


Figure 3 - Simplified conceptualization of the data within the multi-phase stream digital twin

When it comes to the creation of a multiphase stream object in the constructor of the class the following operations are performed as shown in Figure 4; the first operation is to get the datapoints form CDP, these are in the unit tons/h, which is divided by 1000 to get the rates into kg pr hour, and from this point both the gas and water mass rates are accumulated. But the oil has a simplified flash calculation where a mass fraction of the oil is split form the oil mass stream, and called gas in oil (gio), this gas is then accumulated and added to the accumulated gas, to form the gas at separator conditions. The rest of the oil mass rate after the gas portion is removed is accumulated just as the other phases. Figure 5 shows a table from a process simulator simulating flashing with different pressures at multi-phase meter and separator conditions, the rest of the results are in the appendix of collection of non-public documents.

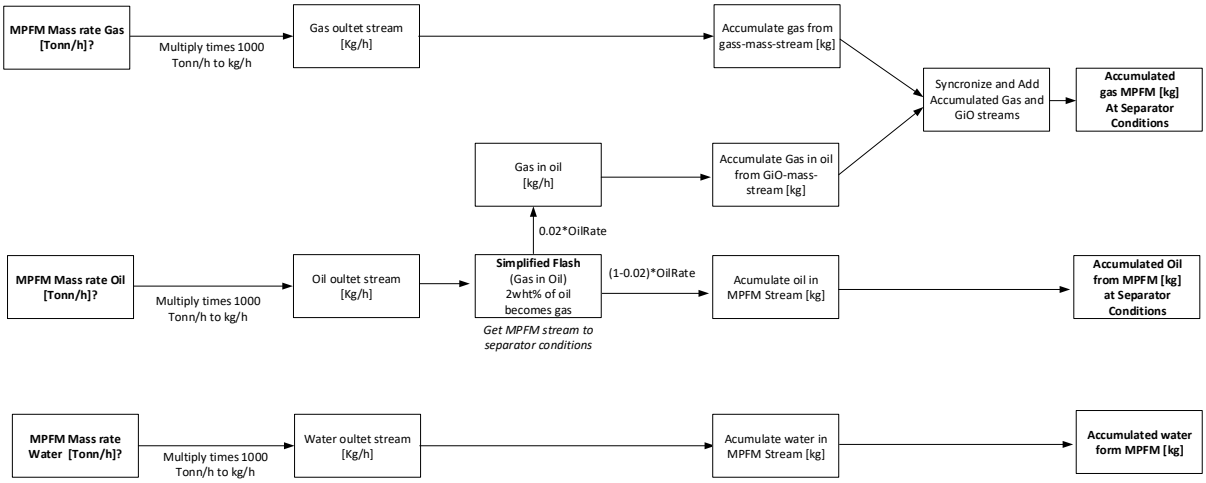


Figure 4 - Multi phase stream calculation flow

Sum of MPFM - SEP Delta Gas std flow	Column Labels										
Row Labels	1500	1600	1700	1800	1900	2000	2100	2200	2300	2400	2500
1500	0.00%	0.58%	1.15%	1.71%	2.25%	2.79%	3.32%	3.84%	4.36%	4.87%	5.37%
1600	-0.58%	0.00%	0.56%	1.12%	1.66%	2.19%	2.72%	3.24%	3.75%	4.26%	4.77%
1700	-1.14%	-0.56%	0.00%	0.55%	1.09%	1.62%	2.14%	2.66%	3.17%	3.68%	4.18%
1800	-1.68%	-1.11%	-0.55%	0.00%	0.54%	1.07%	1.59%	2.10%	2.61%	3.11%	3.61%
1900	-2.20%	-1.63%	-1.08%	-0.53%	0.00%	0.53%	1.04%	1.55%	2.06%	2.56%	3.06%
2000	-2.72%	-2.15%	-1.60%	-1.05%	-0.52%	0.00%	0.52%	1.02%	1.53%	2.03%	2.52%
2100	-3.21%	-2.65%	-2.10%	-1.56%	-1.03%	-0.51%	0.00%	0.51%	1.01%	1.50%	1.99%
2200	-3.70%	-3.14%	-2.59%	-2.06%	-1.53%	-1.01%	-0.50%	0.00%	0.50%	0.99%	1.48%
2300	-4.18%	-3.62%	-3.08%	-2.54%	-2.02%	-1.51%	-1.00%	-0.50%	0.00%	0.49%	0.98%
2400	-4.65%	-4.09%	-3.55%	-3.02%	-2.50%	-1.99%	-1.48%	-0.98%	-0.49%	0.00%	0.48%
2500	-5.11%	-4.56%	-4.02%	-3.49%	-2.97%	-2.46%	-1.96%	-1.46%	-0.97%	-0.48%	0.00%

Figure 5 - Flashing pressure sensitivity

4.2 Separators Streams

The multiphase stream object considers the influent streams, but the single-phase effluent streams are also a crucial part of the data required to perform a calibration.

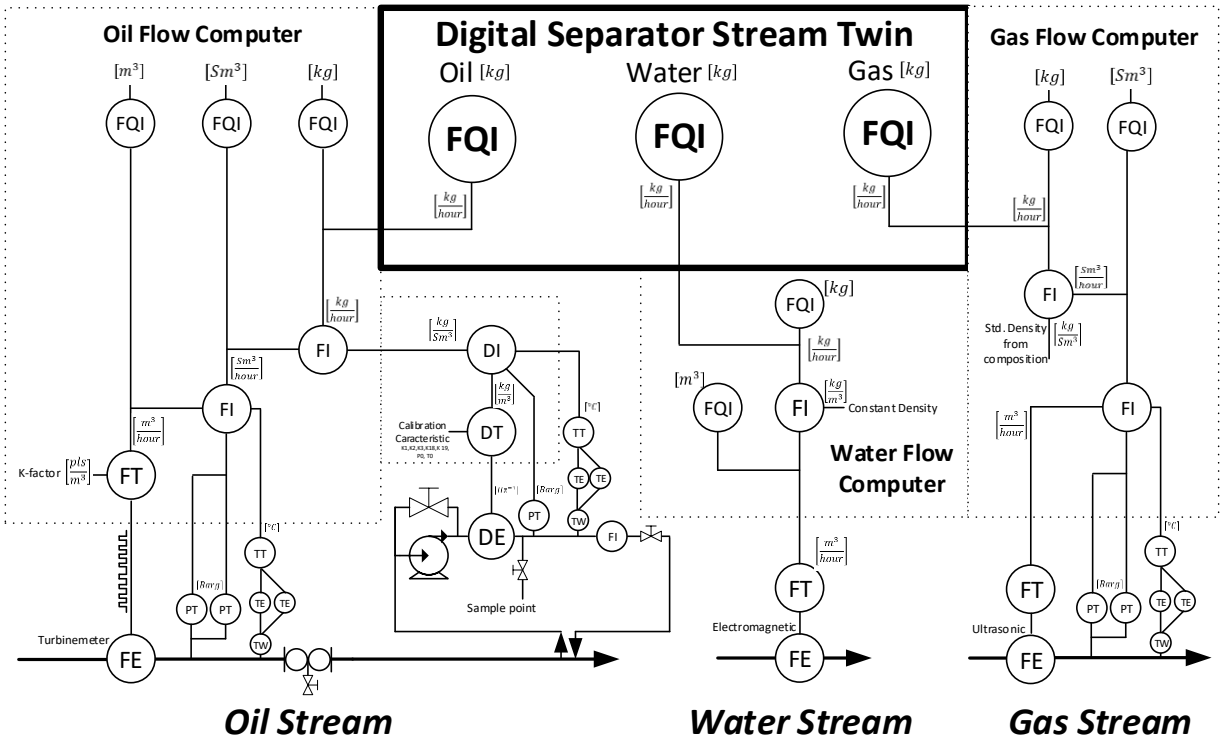


Figure 6 – Simplified conceptualization of the data within the Separator stream digital twin

The separator streams from the data located form the data fusion repository is the standard volumetric flow of gas, the volumetric and mass flow of oil, the volumetric water in oil fraction (water cut) and the volume flow of produced water, at the time of the creation of this method only these values are available, and to be able to convert the volumetric gas flow in to a mass flow an standard density of the gas has to be assumed based. And there is also an addition of the water in the oil stream (wio) this is found through the volumetric flow of oil multiplied by the measured volumetric water cut at each instance. Which forms the volumetric water in oil stream this intern like the volumetric water stream is multiplied with a constant density based on laboratory values. These two water streams are accumulated by them self, and added together afterwards. But luckily the available oil mass stream of pure oil is available, and the only conversion done is a multiplication of a thousand go change the unit from ton/hour to kg/hour

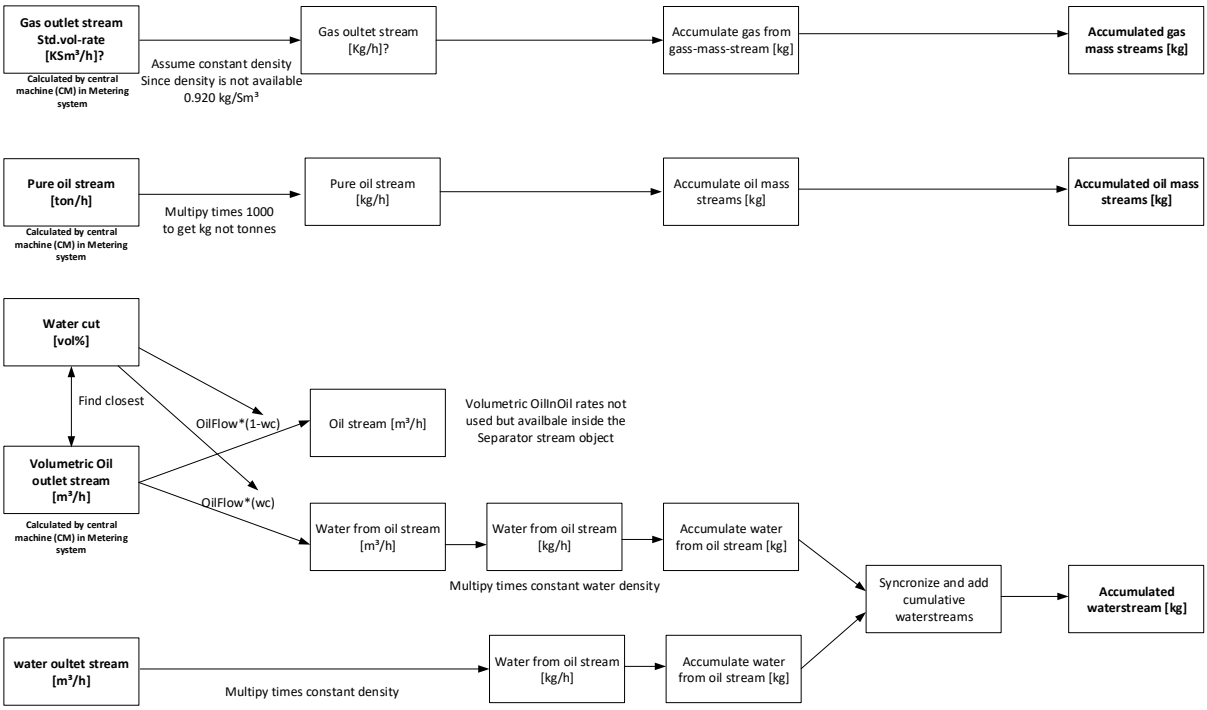


Figure 7 - Separator stream calculation flow

4.3 Trial creation and object orientation

A overview of the classes created is shown in the class diagram in Figure 8, and a typical instantiation of objects form the classes is shown in the object diagram in Figure 9. And the most essential data set in a structure of a trial is shown in Figure 10.

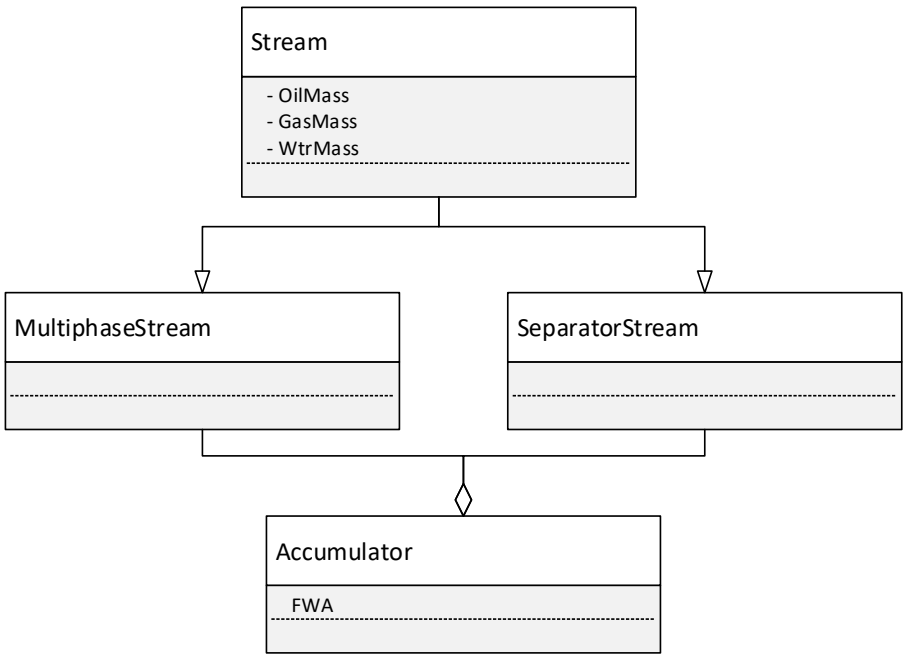


Figure 8 - Class diagram

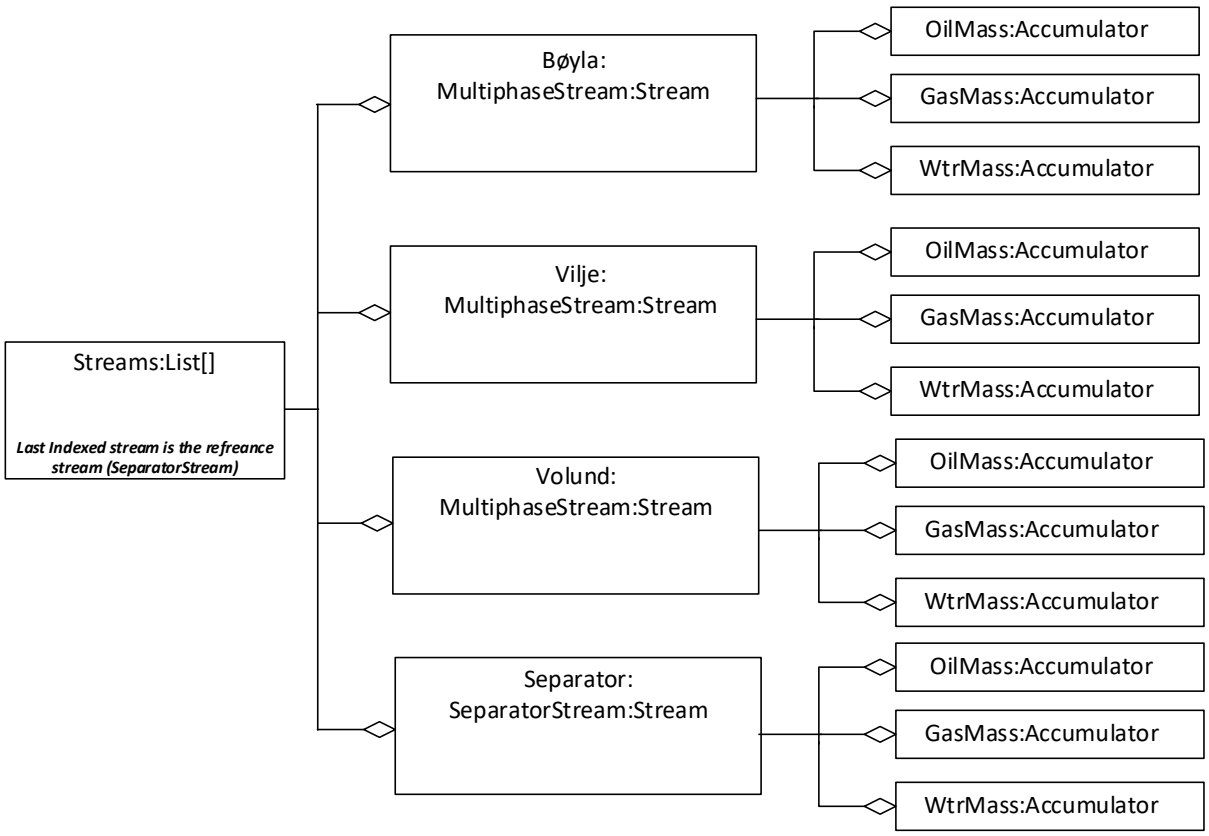


Figure 9 – Typical use of a list of instantiated stream objects in a Object diagram

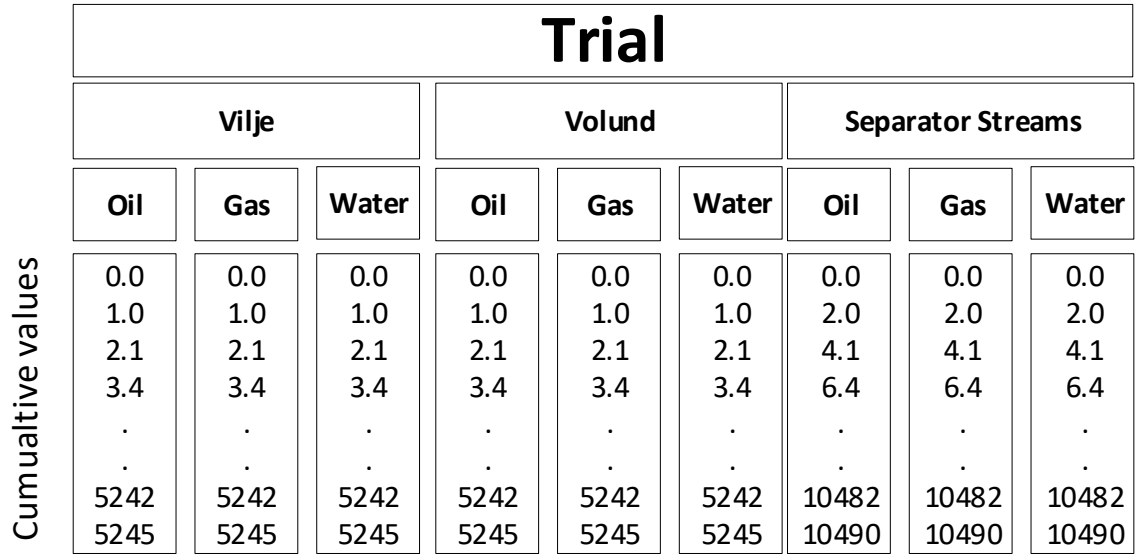


Figure 10 - Example of how cumulative values are set up within a list of streams

4.4 Datapoints toolbox

In the development and testing of code, some of the methods of a more static nature towards calculations of values, indexing / synchronizing datapoints where made and set into its own python file[5] called Datapoints toolbox and contain more general-purpose static methods used by the classes, and also contain code towards the initial testing and creating own traditional calibration method.

4.5 Trial time window locator

There is also a Trial time windows locator which has as an input a search area which is the days where a calibration was performed, then it gets the datapoints from the limit-switches on valves that redirects flow either to the third party separator or to the Alvheim inlet separator. And through basic operations on the received dataframe a time windows for each of the located streams are returned.

[5] Python files have .py as their file extension.

Source code

Following this document is source code related to content of this appendix

Source code – Accumulator

Source code – Streams

Source code – Datapoint toolbox

5 Source code – Accumulator

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 17 18:00:07 2019

@author: Stig
"""

import numpy as np
import pandas as pd
from DatapointsToolbox import *


class Accumulator:
    """
    This is a acumulator of datapoints gotten from the CDP, it also assumes the rate to a
      ccumulate is in pr/hour rates.

    The acumulator has an object inside called data which is a pandas dataframe, and it i
      s not a derived class of the dataframe. though it whould be more elegant and better
       to work with..


    """

    def __init__(self,df,**IntVars):
        """
        the param df is the return object of CogniteClient.datapoints.get_datapoints('tag
      ').to_pandas()
        spesifically - list(stable.datapoints.DatapointsResponse).to_pandas()

        This Performce manly the acumulation process, and adds a 'cumulative' column whic
      h acumulates the data.

        ATM it converts a pr Hour rate.

        """
        #df = datapoints_pd_cdp
        df['dt'] = df['timestamp'].diff()/1000 # create dt collumn and converted to sec
        df.dropna(inplace=True)  #Inplace = True, withuth it it copys the df, memory effi
      cient and stability. Also prompting warnings

        """ Trapezoidal method """
        dt = df['dt'].values
        df['PrSecRate'] = (df['value']/60)/60 #Creates a Per secondt rate
        x = df['PrSecRate'].values

        cumulative = np.zeros(len(x))
        increments = np.zeros(len(x))
        increments[0] = x[0]*dt[0] #This value is not used in the accumulation but it kee
      ps the incremental values consistent and has not a zeros at the start
        for i in range(1,len(x)):
            #Trapezoidal increments - The area under the "curve" section
            increments[i]=((x[i-1]+x[i])/2)*dt[i]
            #Summing the increments - Summing the areas of all the curve sections
            cumulative[i] = cumulative[i-1]+((x[i-1]+x[i])/2)*dt[i]

             #Trapezoidal method
        """ End Trapezoidal method"""


```

```
54.
55.     #        df['incr'] = df['value']/(60*60)*df['dt'] #Rate from pr hour to pr sec, and mult
            ipise times dt
56.     #
57.     #        cumulative = np.zeros(len(df))
58.     #
59.     #        #Acumulating increments
60.     #        cumulative[0] = df.loc[1,'incr']
61.     #        for i in range(1,len(cumulative)):
62.     #            cumulative[i] = cumulative[i-
            1] + df.loc[i+1,'incr'] #DF is indexed from 1 np array from 1.
63.     #
64.     #        #Increas accuracy by implementing the trapeziodal method?
65.     #
66.     #        df_cumulative = pd.DataFrame(cumulative,columns=['cumulative'])
67.     #        df_cumulative.index = df_cumulative.index + 1 #
68.     #        df = pd.concat([df,df_cumulative],axis=1)
69.     #
70.
71.             df['incr'] = increments
72.             df['cumulative'] = cumulative
73.
74.             df = df.reset_index()
75.             df = df.drop('index',axis=1) #reset so index starts at 0
76.
77.             df['datetime'] = pd.to_datetime(df['timestamp'], unit='ms') # Create a datetime c
            olumn of timestamp
78.             df['elapsed'] = df['dt'].cumsum() #Elapsed seconds since start of accumulation
79.
80.             self.data = df
81.
82.             #Adding the flow weighted average values
83.             if(len(IntVars)!=0):
84.                 for IntVarName, IntVar in IntVars.items(): #Split the following key value par
            i. the first in the tuple is the name of the key, the second is the value
85.                     FWA(IntVar,IntVarName)
86.
87.
88.
89.
90.         def df(self):
91.             #A simple method returning the dataframe to simplify development.
92.             return self.data
93.
94.         def FWA(self, IntVar, IntVarUnitName):
95.             """
96.             This methdo adds a Extensive variable and creates a represenatble flow weighted a
            verage value for the start to the elapsed value.
97.
98.             It acheves this by first finding the closes datapoint between the acumulated rate
             variable then calculates the extensive variable towards that rate
99.
100.            """
101.            ClosesIndex = Vfindlowestindex(self.data,IntVar)
102.            IntVariable = IntVar['value'].values
103.            incr = self.data['incr'].values
104.            cumulative = self.data['cumulative'].values
105.            FWA = np.zeros(len(incr))
106.            FWA[0] =IntVariable[0]
107.            for i in range(1,len(FWA)):
108.                FWA[i] = FWA[i-
            1]+(incr[i]*IntVariable[ClosesIndex[i]]) #Buildning the numerator of the FWA
109.
110.            FWA = np.divide(FWA,cumulative) #Deviding the numerator by the denomenator witch
            is the cumulative untill a point.
111.
```

```
112.            self.data[IntVarUnitName] = FWA
113.
114.
115.        def findIndexToAdd(Main,Added):
116.            """
117.            Function used when syncronicing the elapstime between two streams to be added to
              eachother
118.            The first Acumulator added becomes the main witch the other timestamps are syncro
              nized to.
119.            returns a np.array addedIndexes to the added.
120.
121.            """
122.            AddedIndex = np.zeros(len(Main)).astype(int)
123.
124.            MainElapsed = Main['elapsed'].values#.astype(int)
125.            AddedElapsed = Added['elapsed'].values#.astype(int)
126.            #Find the closest elapsed time, to add acumulated at spesific time.
127.
128.            for i in range(1,len(AddedIndex)):
129.
130.                DT = np.zeros(len(AddedElapsed))
131.                for j in range(len(AddedElapsed)):
132.                    DT[j] = abs(AddedElapsed[j]-MainElapsed[i])
133.                res = np.where(DT == min(DT))
134.                AddedIndex[i] = int(max(res[0]))
135.
136.            return AddedIndex
137.
138.        def AddStreams(Main,ToBeAdded):
139.            """
140.            Adding to accumulator streams by syncronizing elapsed times and adds one to anoth
              er
141.            returns a np array of the syncronzed and added cumulative streams
142.            """
143.            TBAIndecies = findIndexToAdd(Main,ToBeAdded)
144.            MainCumulative = Main['cumulative'].values
145.            ToBeAddedToMain = ToBeAdded['cumulative'].values
146.
147.            for i in range(len(TBAIndecies-1)):
148.                MainCumulative[i] = MainCumulative[i] + ToBeAddedToMain[TBAIndecies[i]]
149.
150.            return MainCumulative
151.
152.        def AddStream(self,Accumulator):
153.            self.data['cumulative'] = AddStreams(self.data,Accumulator)
154.
155.
156.
157.
158.    # "STATIC FUNCTIONS"   ??
159.
160.    def findIndexToAdd(Main,Added):
161.        """
162.        Function used when syncronicing the elapstime between two streams to be added to each
              other
163.        The first Acumulator added becomes the main witch the other timestamps are syncronize
              d to.
164.        returns a np.array addedIndexes to the added.
165.
166.        """
167.        AddedIndex = np.zeros(len(Main)).astype(int)
168.
169.        MainElapsed = Main['elapsed'].values#.astype(int)
170.        AddedElapsed = Added['elapsed'].values#.astype(int)
171.        #Find the closest elapsed time, to add acumulated at spesific time.
172.
```

```
        for i in range(1,len(AddedIndex)):

            DT = np.zeros(len(AddedElapsed))
            for j in range(len(AddedElapsed)):
                DT[j] = abs(AddedElapsed[j]-MainElapsed[i])
            res = np.where(DT == min(DT))
            AddedIndex[i] = int(max(res[0]))

        return AddedIndex

    def AddStreams(Main,ToBeAdded):
        """
        Adding to accumulator streams by syncronizing elapsed times and adds one to another
        returns a np array of the syncronzed and added cumulative streams
        """
        TBAIndecies = findIndexToAdd(Main,ToBeAdded)
        MainCumulative = Main['cumulative'].values
        ToBeAddedToMain = ToBeAdded['cumulative'].values

        for i in range(len(TBAIndecies-1)):
            MainCumulative[i] = MainCumulative[i] + ToBeAddedToMain[TBAIndecies[i]]

        return MainCumulative


    if (__name__ == '__main__'):
        import matplotlib.pyplot as plt

        #Unit testing of Accumulator
        hours = 4
        t = np.linspace(0,1000,1001)*1000
        tr = np.copy(t)
        for i in range(len(tr)):
            tr[i] = tr[i]+i*10000+(np.random.rand()-0.5)*5000
        div = 5*(np.random.rand(len(t))-0.5) #Give values som randoms to accumulate
        X = np.zeros(len(t))
        X[0] = 150
        for i in range(1,len(t)):
            X[i]=X[i-1]+div[i]

        plt.plot(X)
        plt.show()
        df = pd.DataFrame({'timestamp':tr,'value':X})
        Test = Accumulator(df)
        plt.plot(Test.data['cumulative'])
        plt.show()
        print(Test.data.head(20))

        print('---- Unit test --- of accumulator')
        X = np.ones(len(t))*60*60

        plt.plot(X)
        plt.show()
        df_check = pd.DataFrame({'timestamp':t,'value':X})
        UnitTest = Accumulator(df_check)
        plt.plot(UnitTest.data['cumulative'])
        print(UnitTest.data['cumulative'].tail(20))
```

6 Source code – Streams

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 17 18:00:07 2019

@author: Stig
"""

import numpy as np
import pandas as pd
from DatapointsToolbox import *


class Accumulator:
    """
    This is a acumulator of datapoints gotten from the CDP, it also assumes the rate to accumulate is in pr/hour rates.

    The acumulator has an object inside called data which is a pandas dataframe, and it is not a derived class of the dataframe. though it whould be more elegant and better to work with..


    """

    def __init__(self,df,**IntVars):
        """
        the param df is the return object of CogniteClient.datapoints.get_datapoints('tag').to_pandas()
        spesifically - list(stable.datapoints.DatapointsResponse).to_pandas()

        This Performce manly the acumulation process, and adds a 'cumulative' column which acumulates the data.

        ATM it converts a pr Hour rate.

        """
        #df = datapoints_pd_cdp
        df['dt'] = df['timestamp'].diff()/1000 # create dt collumn and converted to sec

        df.dropna(inplace=True)  #Inplace = True, withuth it it copys the df, memory efficient and stability. Also prompting warnings

        """ Trapezoidal method """
        dt = df['dt'].values
        df['PrSecRate'] = (df['value']/60)/60 #Creates a Per secondt rate
        x = df['PrSecRate'].values

        cumulative = np.zeros(len(x))
        increments = np.zeros(len(x))
        increments[0] = x[0]*dt[0] #This value is not used in the accumulation but it keeps the incremental values consistent and has not a zeros at the start
        for i in range(1,len(x)):
            #Trapezoidal increments - The area under the "curve" section
            increments[i]=((x[i-1]+x[i])/2)*dt[i]
            #Summing the increments - Summing the areas of all the curve sections
            cumulative[i] = cumulative[i-1]+((x[i-1]+x[i])/2)*dt[i]

             #Trapezoidal method
        """ End Trapezoidal method"""

```

```
53.
54.
55.     #         df['incr'] = df['value']/(60*60)*df['dt'] #Rate from pr hour to pr sec, and mul
        tipise times dt
56.     #
57.     #         cumulative = np.zeros(len(df))
58.     #
59.     #         #Acumulating increments
60.     #         cumulative[0] = df.loc[1,'incr']
61.     #         for i in range(1,len(cumulative)):
62.     #             cumulative[i] = cumulative[i-
        1] + df.loc[i+1,'incr'] #DF is indexed from 1 np array from 1.
63.     #
64.     #         #Increas accuracy by implementing the trapeziodal method?
65.     #
66.     #         df_cumulative = pd.DataFrame(cumulative,columns=['cumulative'])
67.     #         df_cumulative.index = df_cumulative.index + 1 #
68.     #         df = pd.concat([df,df_cumulative],axis=1)
69.     #
70.
71.             df['incr'] = increments
72.             df['cumulative'] = cumulative
73.
74.             df = df.reset_index()
75.             df = df.drop('index',axis=1) #reset so index starts at 0
76.
77.             df['datetime'] = pd.to_datetime(df['timestamp'], unit='ms') # Create a datetime
        column of timestamp
78.             df['elapsed'] = df['dt'].cumsum() #Elapsed seconds since start of accumulation
79.
80.             self.data = df
81.
82.             #Adding the flow weighted average values
83.             if(len(IntVars)!=0):
84.                 for IntVarName, IntVar in IntVars.items(): #Split the following key value pa
        ri. the first in the tuple is the name of the key, the second is the value
85.                     FWA(IntVar,IntVarName)
86.
87.
88.
89.
90.         def df(self):
91.             #A simple method returning the dataframe to simplify development.
92.             return self.data
93.
94.         def FWA(self, IntVar, IntVarUnitName):
95.             """
96.             This methdo adds a Extensive variable and creates a represenatble flow weighted
        average value for the start to the elapsed value.
97.
98.             It acheves this by first finding the closes datapoint between the acumulated rat
        e variable then calculates the extensive variable towards that rate
99.
100.            """
101.            ClosesIndex = Vfindlowestindex(self.data,IntVar)
102.            IntVariable = IntVar['value'].values
103.            incr = self.data['incr'].values
104.            cumulative = self.data['cumulative'].values
105.            FWA = np.zeros(len(incr))
106.            FWA[0] =IntVariable[0]
107.            for i in range(1,len(FWA)):
108.                FWA[i] = FWA[i-
        1]+(incr[i]*IntVariable[ClosesIndex[i]]) #Buildning the numerator of the FWA
109.
110.            FWA = np.divide(FWA,cumulative) #Deviding the numerator by the denomenator witch
         is the cumulative untill a point.
```

```
111.
112.            self.data[IntVarUnitName] = FWA
113.
114.
115.        def findIndexToAdd(Main,Added):
116.            """
117.            Function used when syncronicing the elapstime between two streams to be added to
         eachother
118.            The first Acumulator added becomes the main witch the other timestamps are syncr
        onized to.
119.            returns a np.array addedIndexes to the added.
120.
121.            """
122.            AddedIndex = np.zeros(len(Main)).astype(int)
123.
124.            MainElapsed = Main['elapsed'].values#.astype(int)
125.            AddedElapsed = Added['elapsed'].values#.astype(int)
126.            #Find the closest elapsed time, to add acumulated at spesific time.
127.
128.            for i in range(1,len(AddedIndex)):
129.
130.                DT = np.zeros(len(AddedElapsed))
131.                for j in range(len(AddedElapsed)):
132.                    DT[j] = abs(AddedElapsed[j]-MainElapsed[i])
133.                res = np.where(DT == min(DT))
134.                AddedIndex[i] = int(max(res[0]))
135.
136.            return AddedIndex
137.
138.        def AddStreams(Main,ToBeAdded):
139.            """
140.            Adding to accumulator streams by syncronizing elapsed times and adds one to anot
        her
141.            returns a np array of the syncronzed and added cumulative streams
142.            """
143.            TBAIndecies = findIndexToAdd(Main,ToBeAdded)
144.            MainCumulative = Main['cumulative'].values
145.            ToBeAddedToMain = ToBeAdded['cumulative'].values
146.
147.            for i in range(len(TBAIndecies-1)):
148.                MainCumulative[i] = MainCumulative[i] + ToBeAddedToMain[TBAIndecies[i]]
149.
150.            return MainCumulative
151.
152.        def AddStream(self,Accumulator):
153.            self.data['cumulative'] = AddStreams(self.data,Accumulator)
154.
155.
156.
157.
158.    # "STATIC FUNCTIONS"   ??
159.
160.    def findIndexToAdd(Main,Added):
161.        """
162.        Function used when syncronicing the elapstime between two streams to be added to eac
        hother
163.        The first Acumulator added becomes the main witch the other timestamps are syncroniz
        ed to.
164.        returns a np.array addedIndexes to the added.
165.
166.        """
167.        AddedIndex = np.zeros(len(Main)).astype(int)
168.
169.        MainElapsed = Main['elapsed'].values#.astype(int)
170.        AddedElapsed = Added['elapsed'].values#.astype(int)
171.        #Find the closest elapsed time, to add acumulated at spesific time.
```

```
172.
173.        for i in range(1,len(AddedIndex)):
174.
175.            DT = np.zeros(len(AddedElapsed))
176.            for j in range(len(AddedElapsed)):
177.                DT[j] = abs(AddedElapsed[j]-MainElapsed[i])
178.            res = np.where(DT == min(DT))
179.            AddedIndex[i] = int(max(res[0]))
180.
181.        return AddedIndex
182.
183.    def AddStreams(Main,ToBeAdded):
184.        """
185.        Adding to accumulator streams by syncronizing elapsed times and adds one to another
186.        returns a np array of the syncronzed and added cumulative streams
187.        """
188.        TBAIndecies = findIndexToAdd(Main,ToBeAdded)
189.        MainCumulative = Main['cumulative'].values
190.        ToBeAddedToMain = ToBeAdded['cumulative'].values
191.
192.        for i in range(len(TBAIndecies-1)):
193.            MainCumulative[i] = MainCumulative[i] + ToBeAddedToMain[TBAIndecies[i]]
194.
195.        return MainCumulative
196.
197.
198.    if (__name__ == '__main__'):
199.        import matplotlib.pyplot as plt
200.
201.        #Unit testing of Accumulator
202.        hours = 4
203.        t = np.linspace(0,1000,1001)*1000
204.        tr = np.copy(t)
205.        for i in range(len(tr)):
206.            tr[i] = tr[i]+i*10000+(np.random.rand()-0.5)*5000
207.        div = 5*(np.random.rand(len(t))-0.5) #Give values som randoms to accumulate
208.        X = np.zeros(len(t))
209.        X[0] = 150
210.        for i in range(1,len(t)):
211.            X[i]=X[i-1]+div[i]
212.
213.        plt.plot(X)
214.        plt.show()
215.        df = pd.DataFrame({'timestamp':tr,'value':X})
216.        Test = Accumulator(df)
217.        plt.plot(Test.data['cumulative'])
218.        plt.show()
219.        print(Test.data.head(20))
220.
221.        print('---- Unit test --- of accumulator')
222.        X = np.ones(len(t))*60*60
223.
224.        plt.plot(X)
225.        plt.show()
226.        df_check = pd.DataFrame({'timestamp':t,'value':X})
227.        UnitTest = Accumulator(df_check)
228.        plt.plot(UnitTest.data['cumulative'])
229.        print(UnitTest.data['cumulative'].tail(20))
```

7 Source code – Datapoint toolbox

```
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 28 19:39:54 2019

@author: stig
"""
import numpy as np
import pandas as pd
from datetime import datetime
from datetime import timedelta

def findlowestindex(biggest_df,smallest_df):
    lowestindex = np.zeros(len(biggest_df))  #By lowest index the, meas that the in
dex covertion with the least time between.
    lowestArr = np.zeros(len(biggest_df))

    j_high = 0
    for i in range(len(biggest_df)):
        lowest = np.inf #High initial start value
        j_ref = np.inf #High initial start valyue
        rise = False

        for j in range(len(smallest_df)):
            if(j_high==0):
                jq = j
            else:
                jq = j_high+j #effiecient excecution of the algorithm, no need to l
oop the entire array in each inner loop.
            if(jq>=len(smallest_df)-1): # Out of datapoints, out of datapoint-
timestamps
                break


            DT = abs(smallest_df.loc[jq,'timestamp']-
biggest_df.loc[i,'timestamp'])

            if(DT<lowest):
                lowest = DT
            else:
                rise = True
            if(j_ref >= lowest) and rise:
                lowestindex[i] = jq-1
                lowestArr[i] = lowest
                break #Found closest for this index i iteration.
            J_ref = DT

        j_high = lowestindex[i]

    #check last idexes if timestamp out of range with respect to the other
    if(lowestindex[len(lowestindex)-
1] == 0): #Indicate need by seeing if last index has value 0.
        for i in range(1,len(lowestindex)): #loops through entire array, to ensure
no bugs from odd division of index etc.
            if(lowestindex[i]<lowestindex[i-
1]): #checks to se that each index is same or higher value
                lowestindex[i] = lowestindex[i-
1]   # if not set to same value as before.

    return lowestindex

def Vfindlowestindex(biggest_df,smallest_df):
```

```
54.         lowestindex = np.zeros(len(biggest_df))  #By lowest index the, meas that the in
        dex covertion with the least time between.
55.         lowestArr = np.zeros(len(biggest_df))
56.         biggest = biggest_df['timestamp'].values.astype(int)
57.         smallest = smallest_df['timestamp'].values.astype(int)
58.
59.
60.         j_high = 0
61.         for i in range(len(biggest)):
62.             lowest = np.inf #High initial start value
63.             j_ref = np.inf #High initial start valyue
64.             rise = False
65.
66.             for j in range(len(smallest)):
67.                 if(j_high==0):
68.                     jq = j
69.                 else:
70.                     jq = j_high+j #effiecient excecution of the algorithm, no need to l
        oop the entire array in each inner loop.
71.                 if(jq>=len(smallest)-1): # Out of datapoints, out of datapoint-
        timestamps
72.                     break
73.
74.                 jq= int(jq)
75.
76.                 DT = abs(smallest[jq]-biggest[i])
77.
78.                 if(DT<lowest):
79.                     lowest = DT
80.                 else:
81.                     rise = True
82.                 if(j_ref >= lowest) and rise:
83.                     lowestindex[i] = jq-1
84.                     lowestArr[i] = lowest
85.                     break #Found closest for this index i iteration.
86.                 j_ref = DT
87.
88.             j_high = lowestindex[i]
89.
90.         #check last idexes if timestamp out of range with respect to the other
91.         if(lowestindex[len(lowestindex)-
        1] == 0): #Indicate need by seeing if last index has value 0.
92.             for i in range(1,len(lowestindex)): #loops through entire array, to ensure
        no bugs from odd division of index etc.
93.                 if(lowestindex[i]<lowestindex[i-
        1]): #checks to se that each index is same or higher value
94.                     lowestindex[i] = lowestindex[i-
        1]   # if not set to same value as before.
95.
96.         return lowestindex
97.
98.     def find_nearest(array, value):
99.         array = np.asarray(array)
100.        idx = (np.abs(array - value)).argmin()
101.        return idx
102.
103.    def findIndexToAdd(Main,Added):
104.        AddedIndex = np.zeros(len(Main)).astype(int)
105.
106.        MainElapsed = Main['elapsed'].values#.astype(int)
107.        AddedElapsed = Added['elapsed'].values#.astype(int)
108.        #Find the closest elapsed time, to add acumulated at spesific time.
109.
110.        for i in range(1,len(AddedIndex)):
111.
112.            DT = np.zeros(len(AddedElapsed))
```

```
113.            for j in range(len(AddedElapsed)):
114.                DT[j] = abs(AddedElapsed[j]-MainElapsed[i])
115.            res = np.where(DT == min(DT))
116.            AddedIndex[i] = int(max(res[0]))
117.
118.        return AddedIndex
119.
120.    def AddStreams(Main,ToBeAdded):
121.        """
122.        Adding to accumulator streams by syncronizing elapsed times and adds one to ano
        ther
123.        returns a np array of the syncronzed and added cumulative streams
124.        """
125.        TBAIndecies = findIndexToAdd(Main,ToBeAdded)
126.        MainCumulative = Main['cumulative'].values
127.        ToBeAddedToMain = ToBeAdded['cumulative'].values
128.
129.        for i in range(len(TBAIndecies-1)):
130.            MainCumulative[i] = MainCumulative[i] + ToBeAddedToMain[TBAIndecies[i]]
131.
132.        return MainCumulative
133.
134.
135.    def SyncTimestamp(smallest_df, largest_df):
136.        #Converting ms timestamps to pr 10 sec "[desisec]", and converts them to intege
        r, som now it is find the closes matchin
137.        outer = (smallest_df['timestamp'].values/10000).astype(int)
138.        inner = (largest_df['timestamp'].values/10000).astype(int)
139.        start = 1
140.        pairs = []
141.        for i in range(start,len(outer)-1):
142.            highest = 0
143.
144.            for j in range(start,len(inner)-1):
145.                if(j+highest > len(inner)-1):
146.                    break
147.                else:
148.                    inner_index = j+highest
149.
150.                if(inner[inner_index]==outer[i]):
151.                    pairs.append((i,j))
152.                    highest = j
153.        return pairs
154.
155.    def K_Trad(Subject, Referance):
156.        """
157.        returns a touple of two np.arrays (elapsed , k_factors) where elapsed is the ac
        cumulation time and k_factors
158.        """
159.        #Klist = []
160.
161.        Matchindexes = SyncTimestamp(Subject,Referance)
162.        k_factors = np.zeros(len(Matchindexes))
163.        elapsed = np.zeros(len(Matchindexes))
164.        cnt = 0
165.        for pair in Matchindexes:
166.            i,j = pair
167.            Sub = Subject.loc[i,'cumulative']
168.            Ref = Referance.loc[j,'cumulative']
169.            k_factors[cnt] = Ref/Sub
170.            elapsed[cnt] = Subject.loc[i,'elapsed']/60
171.            cnt = cnt + 1
172.            #Klist.append((timestamp,K_factor))
173.        return (elapsed,k_factors)
```

8 Source code – Time Window locator

Printed 05/05-2019

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  Created on Sat Apr  6 21:31:19 2019
4.
5.  @author: stig
6.  """
7.
8.  import pandas as pd
9.  from datetime import datetime
10.
11. def FindTrialWindows(start,end,cdp, AddMin = 20,RemMin = 120):
12.     #3rd party heade manifold valves
13.     Bøyla = '16HV0210/XGH/PRIM'
14.     Vilje = '16HV0510/XGH/PRIM'
15.     Volund  = '16HV0610/XGH/PRIM'
16.
17.     #inlet separator valves to ensure the multi phase steram is not flow to both the inlet se
          parator and the 3.rd party separator at the same time.
18.
19.     BøylaInlet = '16HV0220/BCH/PRIM'
20.     ViljeInlet = '16HV0520/BCH/PRIM'
21.     VolundInlet = '16HV0620/BCH/PRIM'
22.
23.     ts_name = ['Bøyla', 'Vilje', 'Volund','BøylaInlet','ViljeInlet','VolundInlet']
24.     ts = [Bøyla, Vilje, Volund,BøylaInlet,ViljeInlet,VolundInlet]
25.
26.
27.     #Hent ventil posisjons data fra Cognite over headerventiler til 3rd party separatoren.
28.     df = cdp.datapoints.get_datapoints_frame(time_series=ts,aggregates=["step"], granularity=
          "30second", start=start, end=end)
29.
30.
31.
32.     df['datetime'] = pd.to_datetime(df['timestamp'], unit='ms')
33.
34.     #renaming the 3. party inlet header valves
35.     df.rename(columns = {'16HV0210/XGH/PRIM|stepinterpolation' : ts_name[0], '16HV0510/XGH/PR
          IM|stepinterpolation': ts_name[1],'16HV0610/XGH/PRIM|stepinterpolation':ts_name[2]}
          , inplace = True)
36.
37.     #Remainming the main inlet separator header valves
38.     df.rename(columns = {'16HV0220/BCH/PRIM|stepinterpolation' : ts_name[3], '16HV0520/BCH/PR
          IM|stepinterpolation': ts_name[4],'16HV0620/BCH/PRIM|stepinterpolation':ts_name[5]}
          , inplace = True)
39.
40.
41.
42.     df.interpolate(inplace = True) #Fyller ut NAN'er
43.     df['Paralell'] = df[ts_name[0]]+df[ts_name[1]]+df[ts_name[2]] == 2.0  #Finner tider den s
          om er til Paralell kalibrering.
44.
45.
46.     #Finner tiderne som det er utført parralell kalibrering samt sjekker at linjene ikke står
           til alvheim main inlet separator, at løpa flower kun til 3.part sep.
47.
48.     VilVol_ParTimes = df[df['Paralell']&df['Volund']&df['Vilje']&(df['ViljeInlet']+df['Volund
          Inlet']==0.0)]['timestamp']
49.     BøVol_ParTimes = df[df['Paralell']&df['Bøyla']&df['Volund']&(df['BøylaInlet']+df['VolundI
          nlet']==0.0)]['timestamp']
```

```
50.      BøVil_ParTimes = df[df['Paralell']&df['Bøyla']&df['Vilje']&(df['BøylaInlet']+df['ViljeInl
             et']==0.0)]['timestamp']
51.
52.      # legg til forriggling på at headeren til Alvheim inlet separator også ikke er open, slik
              at en strøm strømmer både til 3.part separator og inlet separator
53.
54.
55.      #Add timedelta
56.      from datetime import timedelta
57.
58.      #adding stabilization time after calibration set in moteon
59.
60.      if(len(VilVol_ParTimes)==0):
61.          print("No Vilje Vound Window found")
62.          ViVoWindow="No Time Window found"
63.      else:
64.          ViVoWindow = (datetime.fromtimestamp(min(VilVol_ParTimes)/1000.0) + timedelta(minutes
              =AddMin) ,datetime.fromtimestamp(max(VilVol_ParTimes)/1000.0)-
              timedelta(minutes=RemMin))
65.      if(len(BøVol_ParTimes)==0):
66.          print("No Bøyla Volund Window found")
67.          BøVoWindow = "No Time Window found"
68.      else:
69.          BøVoWindow = (datetime.fromtimestamp(min(BøVol_ParTimes)/1000.0) + timedelta(minutes=
              AddMin) ,datetime.fromtimestamp(max(BøVol_ParTimes)/1000.0)-
              timedelta(minutes=RemMin))
70.
71.      if(len(BøVil_ParTimes)==0):
72.          print("No Bøyla Vilje Window found")
73.          BøViWindow = "No Time Window found"
74.      else:
75.          BøViWindow = (datetime.fromtimestamp(min(BøVil_ParTimes)/1000.0) + timedelta(minutes=
              AddMin) ,datetime.fromtimestamp(max(BøVil_ParTimes)/1000.0)-
              timedelta(minutes=RemMin))
76.
77.      return (ViVoWindow,BøVoWindow,BøViWindow)
78.
79.
80.  if (__name__ == '__main__'):
81.      #Unit testing of Accumulator
82.      print('test not implemented')
```

1 Appendix D – Parallel Calibration Algorithm

Contents

In the main thesis the method used to calculate the calibration factors is established, but when it comes to the implementation of this method in to an algorithm, and also displaying and plotting the results and performs a statistical analysis on the resulting calibration factors. This Appendix will go into detail about the inner working and synchronization of data as well as the plots and results created and used in this thesis.

1 Appendix D – Parallel Calibration Algorithm 1

Contents 2

2 Development of the algorithm 3

2.1 Scale and synchronize data points 3
2.2 Create the Mp matrix 5
2.2.1 Create Mp Frame 5
2.2.2 Fill Mp Frame 5
2.2.3 Special Case: Traditional through the Parallel Calibration algorithm 7
2.3 Solving the general case 7
2.4 Simplifying execution through object orientation 9
2.5 Asses the quality of a parallel calibration 9
2.6 Store values and recalling the result 13

3 Statistical analysis 14

3.1 Initial autodetection of statistical basis 14
3.2 Calculating random uncertainty of result 14

4 Non-linear solver 15

Source Code 18

5 Source Code – Parallel calibration 19

6 Source Code - Calibration Statistics 40

2 Development of the algorithm

The input to the algorithm is a python list object of Trials, where the Trials are also python list of streams objects as discussed in the previous appendix on the Digital Twins, and in the solving of this algorithm the data layout and calculations done based on values from the digital copies which are used to perform a parallel calibration.

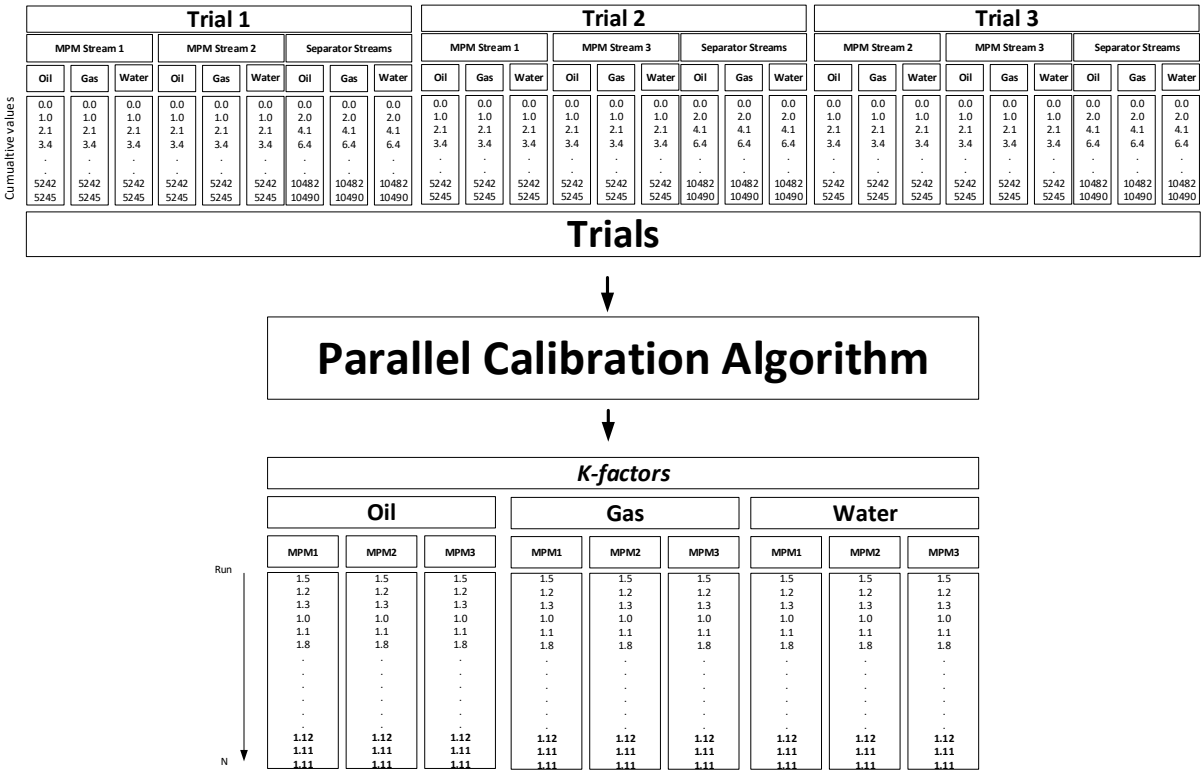


Figure 1 - Input and output of the k-factor calculation within the parallel calibration algorithm, with a 2x2x2 trial configuration.

2.1 Scale and synchronize data points

Each phase in each stream in each trial has their own number of datapoints and when it comes to synchronizing the data points of the cumulative values in each phase stream. This is done by first finding the stream with the lowest number of datapoints in the stream within the time window of the specific trial, this is set as the reference number of runs for the trial. Then each other stream in a give phase is synchronized by finding the closes datapoint to reference datapoint trial and is executed efficiently through elegant use of the NumPy library, as shown in Figure 2.

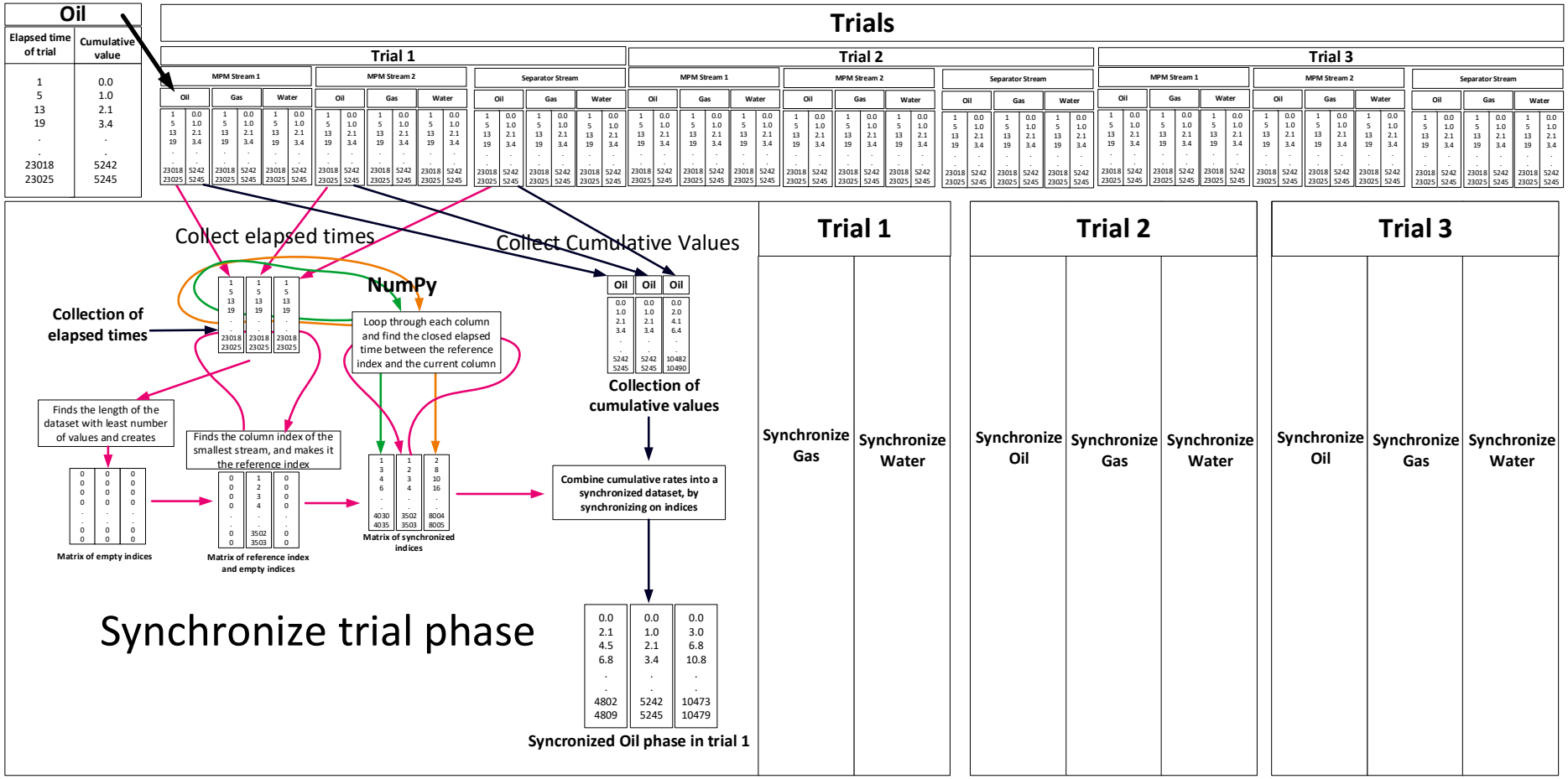


Figure 2 - Figure of the algorithm flow of synchronization of a specific trial phase as the preprocessing before parallel calibration

2.2 Create the M_p matrix

A gold of the parallel calibration is to try to recreate the same developmental k-factors across "time" that the traditional method uses. but when the data comes from T number of different time slots with different elapsed time within the time windows, to look at the x-axis as across time is not necessarily correct which is why the plots don't have time as the x-axis but runs. But in order to create a developmental k-factor across the synchronized trial runs, the structure of the $\boldsymbol{M}_p$ has to be determined, and then later filled with values across the elapsed runs and subsequentially solved to achieve the resulting k-factors.

2.2.1 Create $\boldsymbol{M}_p$ Frame

By looping through all streams in all the trials the algorithm finds through clever lookup of the name of each stream, a frame matrix can be created which can later be filled by values. This initial frame is a binary valued matrix in $\mathbb{R}^{T\times S}$ Dimensions, where a one represents a value to be filled and 0 represents that this stream was not online during this trial.

2.2.2 Fill $\boldsymbol{M}_p$ Frame

When the $\boldsymbol{M}_p$ frame has been established a copy of this frame with float valued cells and this is then to be filled with values for each run and solved. As shown in Figure 3.

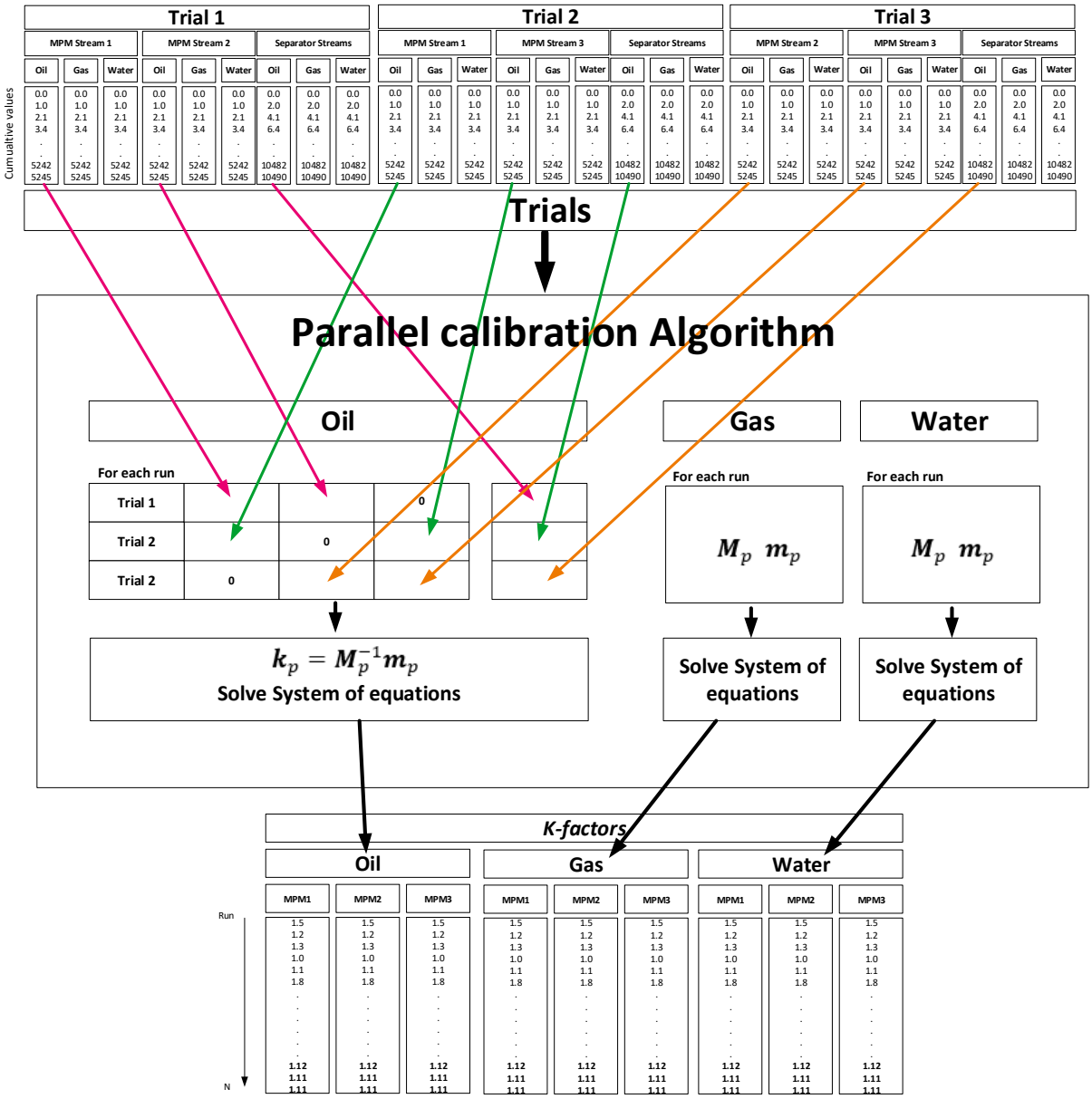


Figure 3 - Inside the algorithm - filling $\boldsymbol{M}_p$ Matrix and $\boldsymbol{m}_p$ vector with a 2x2x2 Trial input

2.2.3 Special Case: Traditional through the Parallel Calibration algorithm

One special case worth mentioning is that if the trial streams given into the parallel calibration algorithm are based purely on a traditional calibration which mean that the only one multiphase stream is entering the separator during a trial. the resulting M_p matrix is a diagonal matrix as shown in equation (2.1) and the invers of a diagonal matrix is the reciprocal of each line on the diagonal as shown in (2.2).

$$\boldsymbol{M}_p \in \mathbb{R}^{T\times S} = \begin{bmatrix} m_{p,1,1} & 0 & \cdots & 0 \\ 0 & m_{p,2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & m_{p,S,\mathrm{T}} \end{bmatrix} \tag{2.1}$$

$$\boldsymbol{M}_p{}^{-1} = \begin{bmatrix} \frac{1}{m_{p,1,1}} & 0 & \cdots & 0 \\ 0 & \frac{1}{m_{p,2,2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{m_{p,S,\mathrm{T}}} \end{bmatrix} \tag{2.2}$$

$$\boldsymbol{M}_p^{-1}\boldsymbol{m}_p = \begin{bmatrix} \frac{1}{m_{p,1,1}} & 0 & \cdots & 0 \\ 0 & \frac{1}{m_{p,2,2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{m_{p,S,\mathrm{T}}} \end{bmatrix} \begin{bmatrix} m_{p,ref,1} \\ m_{p,ref,2} \\ \vdots \\ m_{p,ref,T} \end{bmatrix} = \begin{bmatrix} \left\{\frac{m_{p,ref,1}}{m_{p,1,1}}\right\} \\ \left\{\frac{m_{p,ref,2}}{m_{p,2,2}}\right\} \\ \vdots \\ \left\{\frac{m_{p,ref,T}}{m_{p,S,\mathrm{T}}}\right\} \end{bmatrix} = \begin{bmatrix} k_{p,1} \\ k_{p,2} \\ \vdots \\ k_{p,S} \end{bmatrix} = \mathbf{k}_{\mathrm{p}} \tag{2.3}$$

In solving the system of equation then show that the values received in the $\mathbf{k}_{\mathrm{p}}$vector equation (2.3) are then the same as the values during a traditional calibration.

2.3 Solving the general case

The implementation of the algorithm is developed not to solve a specific case, but the general case with S number of streams, as shown in Figure 4.

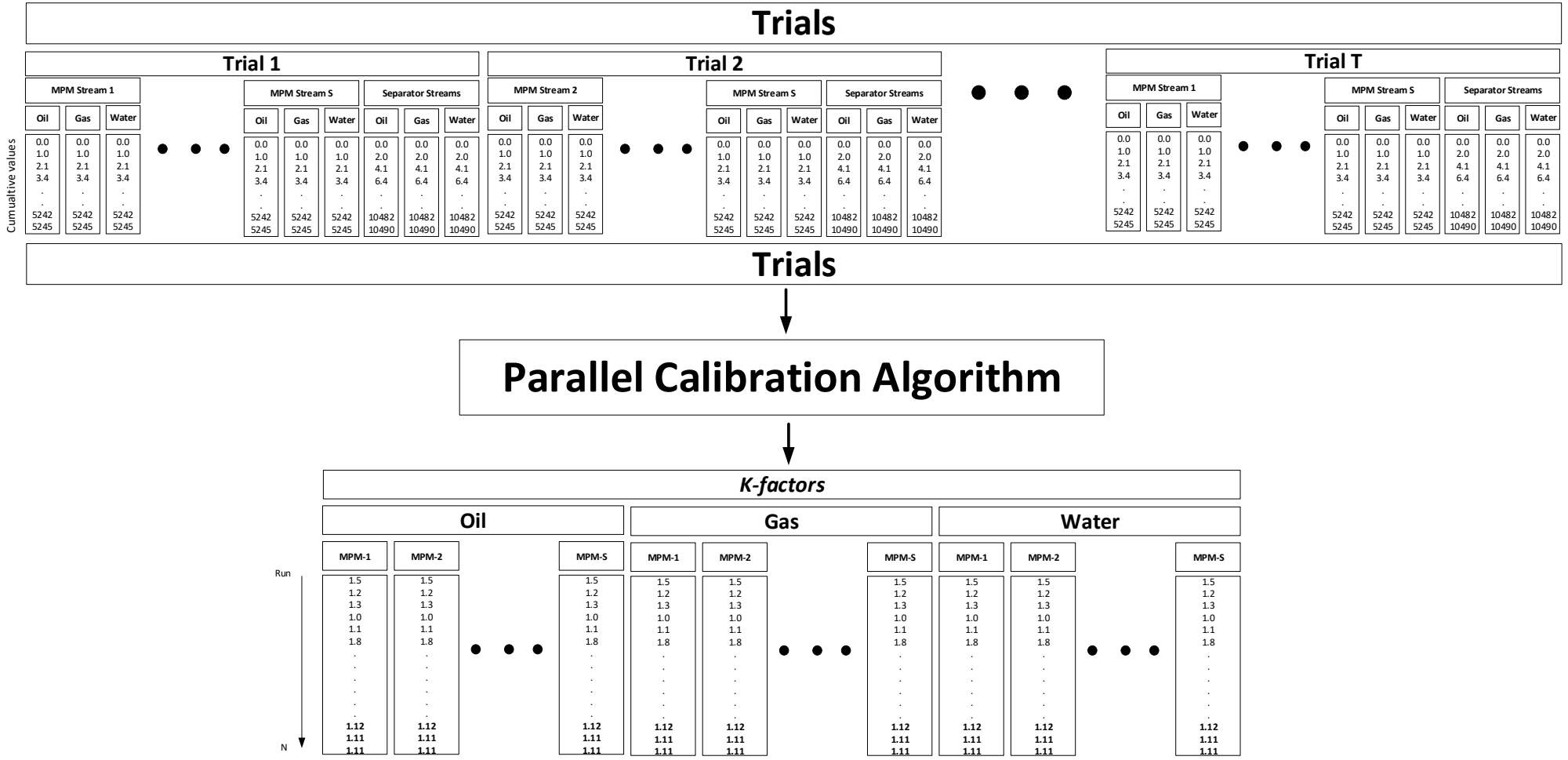


Figure 4 - Data input and output of a general case with S number of streams

2.4 Simplifying execution through object orientation

The parallel calibration method is object oriented in order to simplify the execution of the method and structures the code to a much greater extent, this is to both simplify the storage of the raw data and the creation of a calibration report / data basis. Makes the assessment of the quality of the calibration easier to execute with functions already implemented. As well as all the values from all the trials and the values calculated within the calibration is available within the instantiated parallel calibration object.

2.5 Asses the quality of a parallel calibration

The algorithm contains a structured data source of both aggregated and raw datapoints for the plotting of both the result development, and the conditions during the trial data was created. And a substantial amount of code within the parallel calibration source code is on concerning the control and check the quality / resulting values form the algorithm, as well as inspecting the values within the trials, to ensure that both the input and the output from the algorithm works with and generates representative values to the reality.

But within the parallel calibration object there are the following plots available

- Plot of k-factor development -Figure 5
- Plot of cumulative values in augmented matrix form, which is the raw data which goes into the system of equations - Figure 6
- Plot of the main raw datapoints of the mass flowrates in each phase, which is the raw data collected by each stream object in each trial, and displayed in the augmented matrix form plots - Figure 7
- Plot of the essential states of the separators and the intensive variables in the streams shown in Figure 8.
- Histogram over the time between each datapoint, where the y axis is in log base 10 in order to look into the data quality entering the accumulators, it returns 3 figures one for each phase. Figure 9 show an example of this in the oil phase.

After a statistical analysis there is also two more essential plot available which are;

- A plot containing the calculation basis, with a vertically oriented plot of both a histogram with the number of samples in the x-axis shared with the probability density distribution based on the shown basis. As shown in Figure 10.
- Violin plot of the resulting k-factors for each stream and phase shown in Figure 11

But all data for any further is stored within this calibration object and is available for any data analysis.

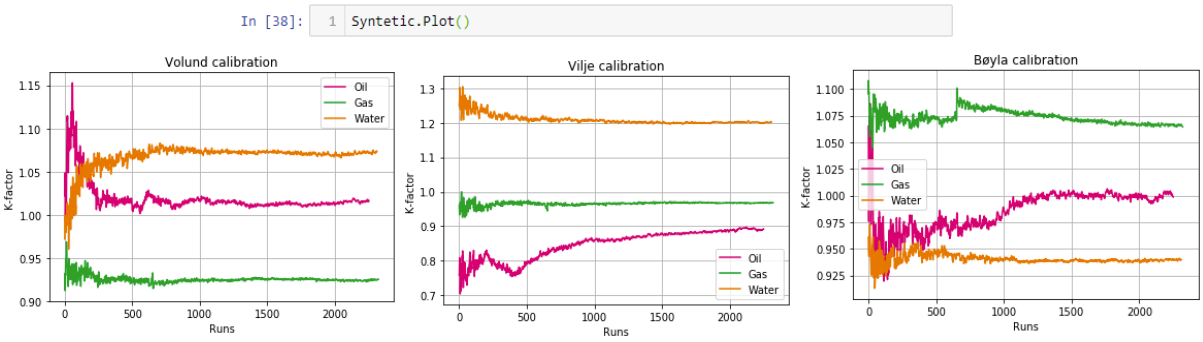


Figure 5 - Plot of resulting k-factor development

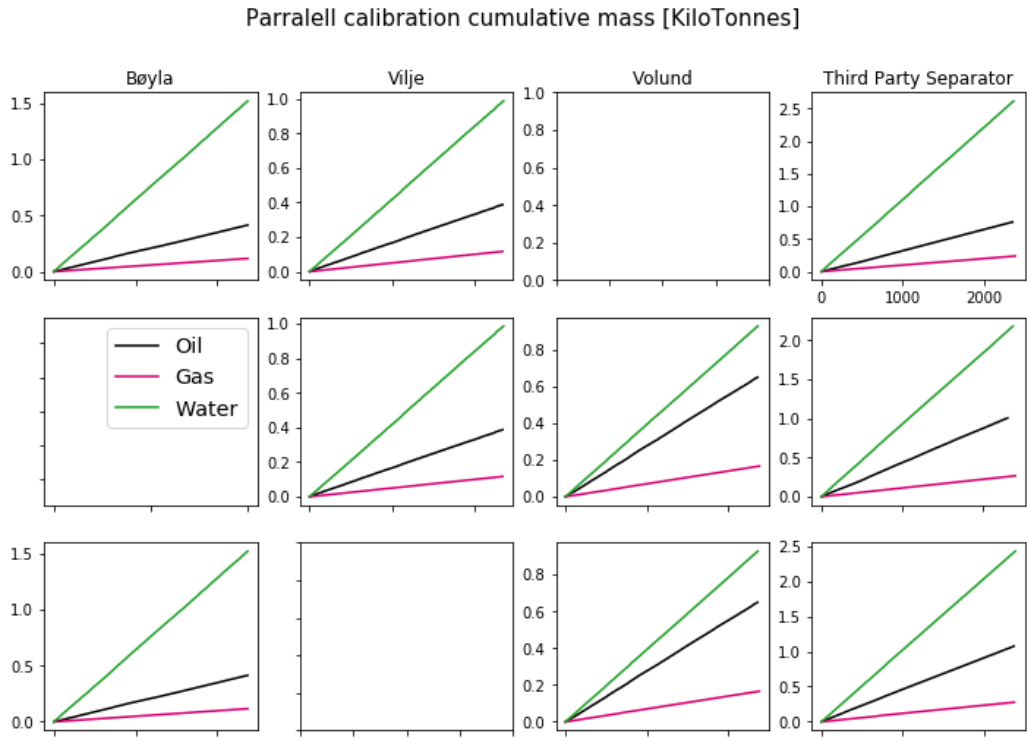


Figure 6 - Plot of the cumulative values in augmented matrix form for a 2x2x2 calibartion

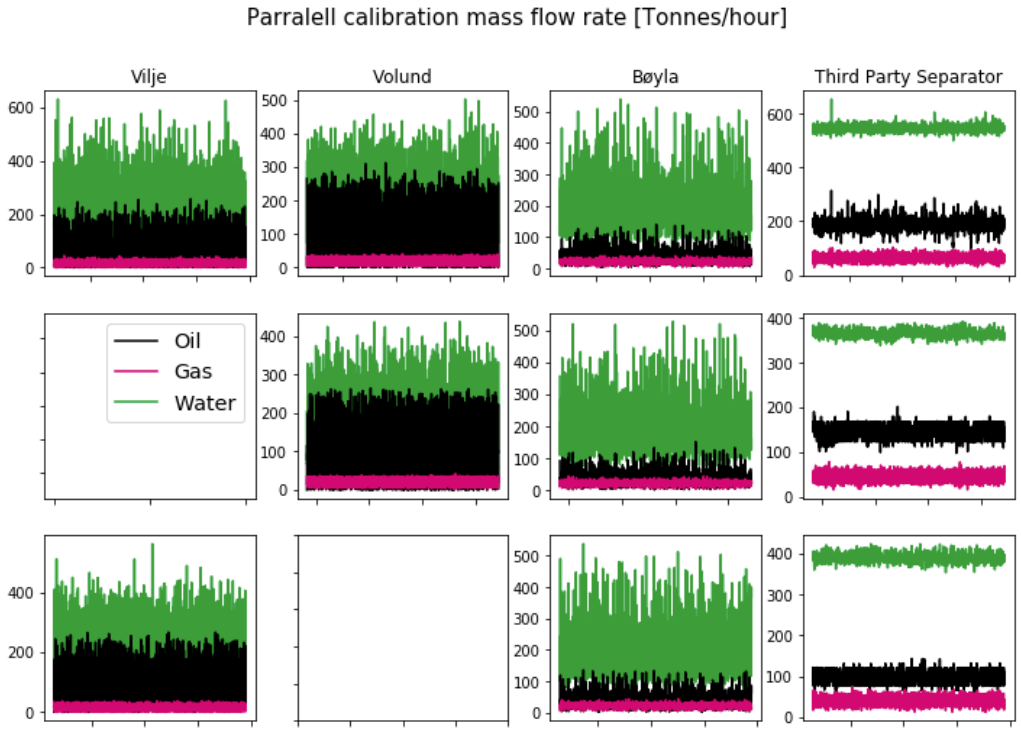


Figure 7 - Plot of phase flowrates in augmented Matrix form for a 3x2x2 calibration.

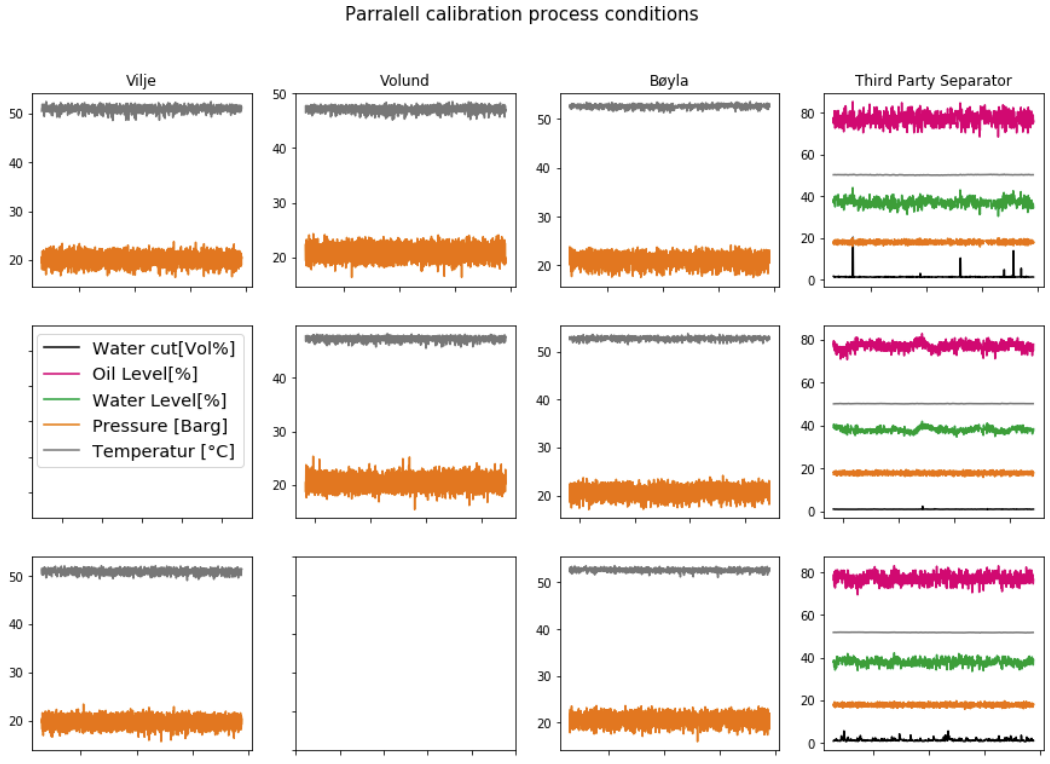


Figure 8 - Plot of intensive variables and relevant states in augmented matrix from for a 3x2x2 calibration

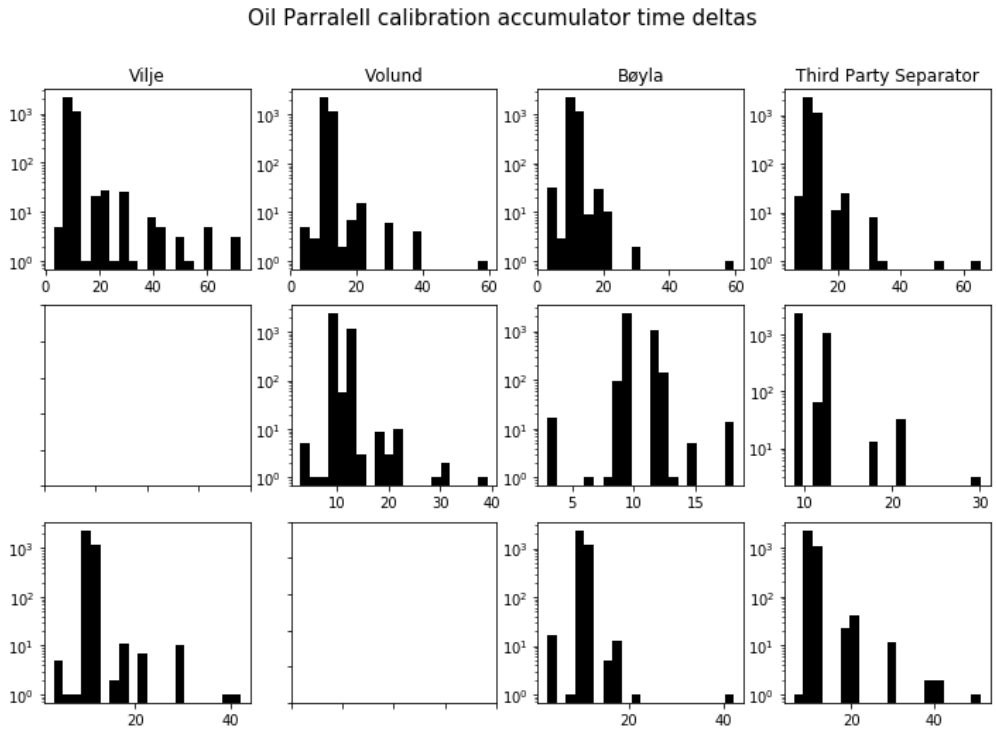


Figure 9 - Histogram Plot of time between each flow rate datapoint, where the y-axis is logarithmically displayed, in augmented matrix form, for a oil parallel calibration

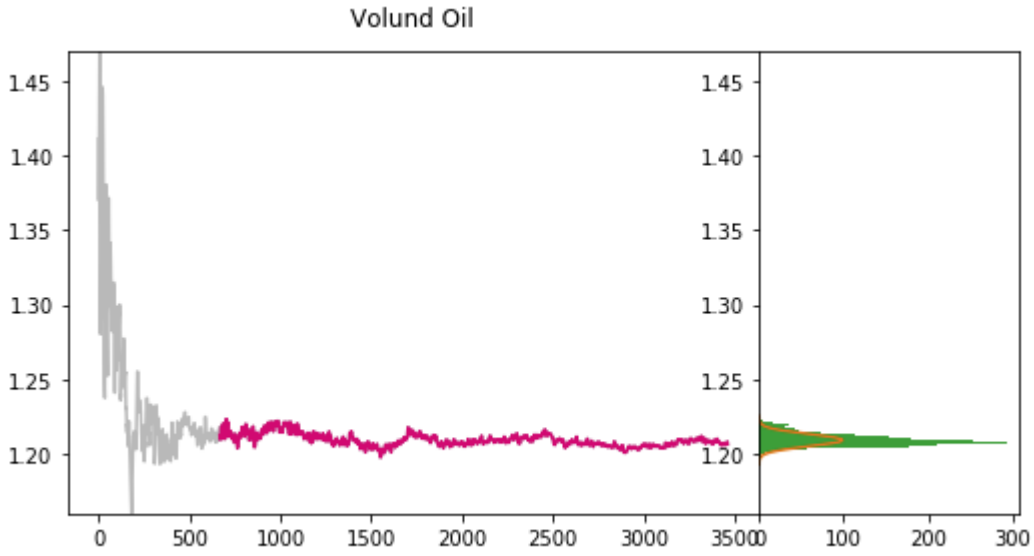


Figure 10 - K-factor basis values for statistical analysis in the left plot and in the right plot a horizontal histogram shared with a normal distribution's probability density plot based on the mean and standard deviation of the basis shown.

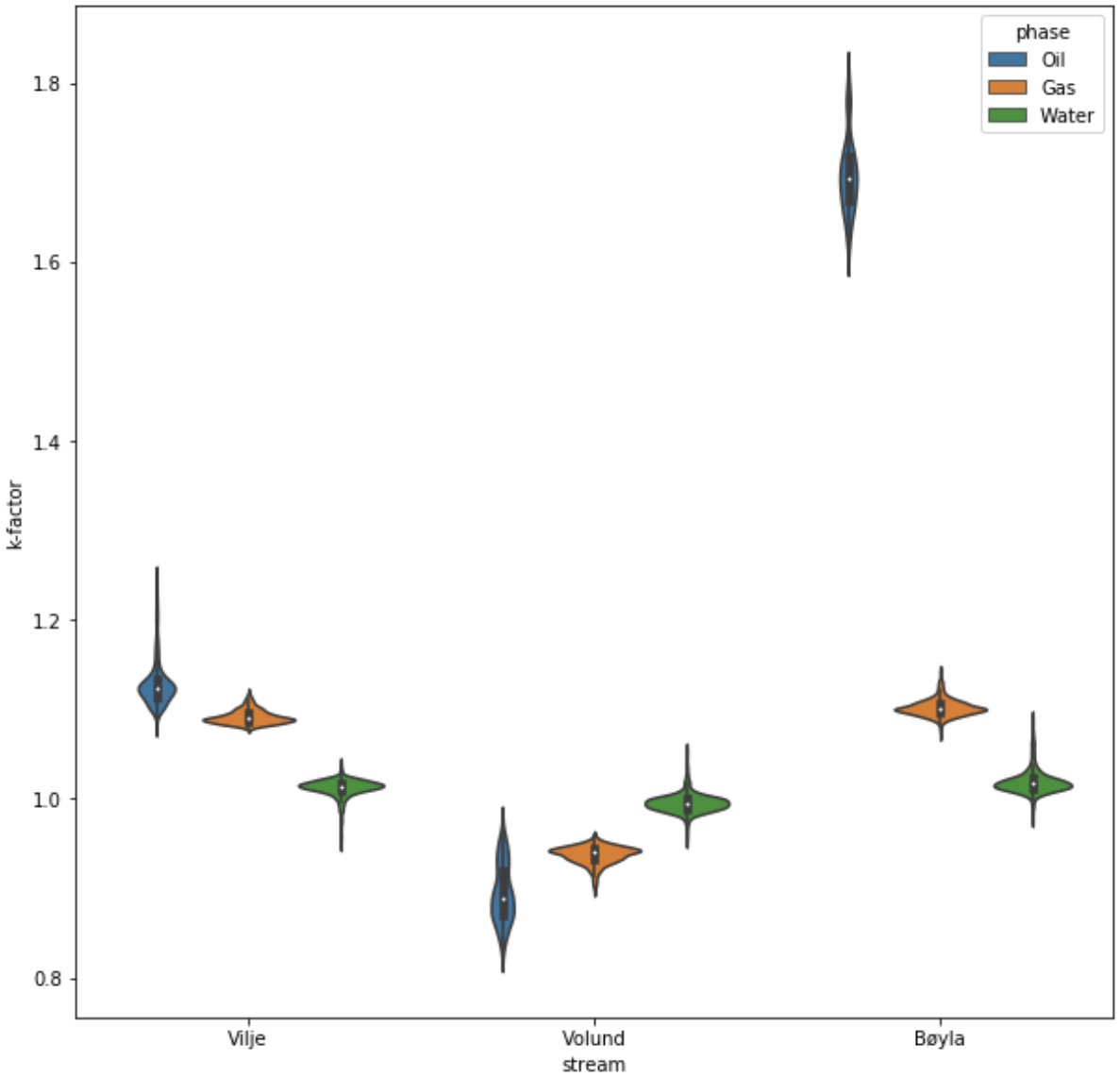


Figure 11 - Violin plot of each streams k-factors for each phase, based on the statistical basis, form a calibration performed in late April 2019

2.6 Store values and recalling the result

When it comes to the storing of results the calibration object instantiated by the parallel calibration class can be serialized through the python library pickle and stored in flatform[1] which can later be recalled and analyzed if needed, additionally all numerical values used are stored in pandas data frames which have inbuilt exporting functions to formats such as .csv [2]and others. Other documentations can also be automatically created by due to limited time an automatic report, typically pdf report is not automatically created, but the Jupyter notebook and or other python interpreters should.

[1] Flat form refers to that the objects in working memory are store to a file in a file system which can later be recovered into the same object.

[2] .csv – file format where columns are separated by a delimiter typically comma ‘,’ or colon ‘;’ rows are separated by end-line character.

3 Statistical analysis

When the method has calculated the K-factors from a collection of trials, each of the k-factors should converge to a stable value, and on a subset of the stabilized k-factors can form a basis data set. This statistical basis dataset can then be used to perform a statistical analysis, on the random uncertainty of the values as well provide a window into the distribution. Assuming this dataset is normally distributed the random uncertainty can be by assessing the normal distribution. A coverage factor of two is used in order to specify the confidence level of the statistical analysis. If the k-factors do not converge and stabilize the dataset of trials used are not good, and new trials needs to be chosen.

3.1 Initial autodetection of statistical basis

The initial part of a statically basis, has implemented a method trying to determine this statistical basis by itself, and it does so by first calculating the sum squared errors (SSE) over a subset between 5 and 90% of the k-factors calculated as shown in equation (3.1).

$$SSE_n = \sum_{i=0.05}^{n} \left(kf_i - \overline{kf}_{0.05\cdot n,n}\right)^2, where\ \overline{kf}_{0.05\cdot n,n} = \frac{1}{n}\sum_{i=0.05\cdot n}^{n} kf_i \tag{3.1}$$

This sub dataset of the sum of the squared errors are then differentiated and squared resulting in a dataset with peaks called the DSSE dataset, this represent changes within the k-factor development (3.2). And a peak detection method form signal library of SciPy called "*find_peaks*" is used with a preset limit, for what a peak could be. This peak detection limit is also a constant of the standard deviation of the dataset used, in order to get a peak within the unique nature of each k-factor development.

$$DSSE_n = (SSE_{n-1} - SSE_n)^2 \tag{3.2}$$

The result of this algorithm is shown in the left plot in Figure 10.

3.2 Calculating random uncertainty of result

Within either the auto detected basis or manually set basis set. A calculation of the random uncertainty is done by assuming the values within the basis are normally distributed, and a numerical integration of the normal distribution's probability density function is performed, with a coverage factor of 2, which is the level of confidence specified in the measurement regulations of the NPD. The numerical integration is done through the trapezoidal method which is the same as the accumulators in the stream classes. The vectorization of the normal distribution is separated into 10000 numerical datapoint within the confidence interval. As shown analytically in equation (3.3), where μ is the mean and σ is the standard deviation of the statistical basis calculated through the NumPy library.

$$p(x|\mu,\sigma) = \int_{\mu-2\sigma}^{\mu+2\sigma} \left[\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right] dx \tag{3.3}$$

All this is shown and displayed if a unit test of the Calibration Statistics code, on a randomly generated dataset.

4 Non-linear solver

Also worth mentioning was a non-linear solver proposed by Torbjørn Selanger which was the initial method developed and planned worth mentioning which a solver was also developed but not implemented into the algorithm, and as the linear solver it solves for one phase at the time, and the following method solves for one phase.

But the non-linear method is based on being able to perform the trial on two and two lines in three trials, and each multiphase meter stream is denoted *A*, *B* and *C* and the separator is denoted ref. And equation (4.1) shows the apparent multi stream k-factor for stream A and B, for a trial. What is also worth noticing here is that the *k* is the reciprocal of the used k-factor.

$$k_{AB} = \frac{\int \dot{m}_A \, dt + \int \dot{m}_B \, dt}{\int \dot{m}_{ref} \, dt} = \frac{m_A + m_B}{m_{ref}} \tag{4.1}$$

And from the balance laws governing the system m_{ref} contain the information of the actual / real / reference sum of the measured which is called $\widetilde{m}_A$and $\widetilde{m}_B$, as shown in equation (4.2) which is inserted into equation (4.3)

$$m_{ref} = \widetilde{m}_A + \widetilde{m}_B \tag{4.2}$$

$$k_{AB} = \frac{m_A + m_B}{m_{ref}} = \frac{m_A}{\widetilde{m}_A + \widetilde{m}_B} + \frac{m_B}{\widetilde{m}_A + \widetilde{m}_B} \tag{4.3}$$

And through algebraic manipulation of (4.3) the reciprocal specific k-factor for each stream has appeared in equation (4.4)

$$k_{AB} = \frac{1}{\frac{\widetilde{m}_A}{m_A} + \frac{\widetilde{m}_B}{m_A}} + \frac{1}{\frac{\widetilde{m}_B}{m_B} + \frac{\widetilde{m}_A}{m_B}} = \frac{1}{\frac{1}{K_A} + \frac{\widetilde{m}_B}{m_A}} + \frac{1}{\frac{1}{K_B} + \frac{\widetilde{m}_A}{m_B}} \tag{4.4}$$

And further through algebraic manipulations the specific k-factor for a meter can be calculated through equation (4.5). but it still contains some of the intrinsic values of m_{ref}

$$K_A = \frac{1}{\left(k_{AB} - \frac{1}{\frac{1}{K_B} + \frac{\widetilde{m}_A}{m_B}}\right)^{-1} + \frac{\widetilde{m}_B}{m_A}} \tag{4.5}$$

These m_{ref} with the tilde mark, are basically the corrected values of the multiphase meter. And the values are realized through the value the method is supposed to solve for.

$$m_{ref} = \widetilde{m}_A + \widetilde{m}_B = m_A k_A + m_B k_B \rightarrow \frac{m_A}{k_A} = \widetilde{m}_A, \frac{m_B}{k_B} = \widetilde{m}_B \tag{4.6}$$

And by using the measured values in (4.6) and inserting them into (4.5)

$$K_A = \frac{1}{\left(k_{AB} - \frac{1}{\frac{1}{K_B} + \frac{\left(\frac{m_A}{k_A}\right)}{m_B}}\right)^{-1} + \frac{(m_{ref} - \tilde{m}_A)}{m_A}} = \frac{1}{\left(k_{AB} - \frac{1}{\frac{1}{K_B} + \frac{1}{k_A}\frac{m_A}{m_B}}\right)^{-1} + \frac{m_{ref}}{m_A} - \frac{1}{K_A}} \quad (4.7)$$

And through the final manipulation of the variables an equation with just a single unknown / non measured variable has been achieved as shown in equation (4.7)

But for the order the measured values going into the method are turned into parameters for the non-linear solver equation 4.8 shows the parameters used in each function.

$$\begin{aligned} \alpha_{AB} = k_{AB} = \frac{m_A + m_B}{m_{ref}} &= \frac{\int \dot{m}_A dt + \int \dot{m}_B dt}{\int \dot{m}_{ref} dt} \\ \beta_{AB} = = \frac{m_A}{m_B} &= \frac{\int \dot{m}_A \, dt}{\int \dot{m}_B \, dt} \\ \gamma_{AB} = \frac{m_{ref}}{m_A} &= \frac{\int \dot{m}_{ref} \, dt}{\int \dot{m}_A \, dt} \end{aligned} \quad (4.8)$$

Then the final equation for an instance is solved all three combinations of values, where the k-factor is the only unknow as shown for one instance in equation (4.9).

$$K_A = \frac{1}{\left(\alpha_{AB} - \frac{1}{\frac{1}{K_B} + \frac{1}{k_A}\beta_{AB}}\right)^{-1} + \gamma_{AB} - \frac{1}{K_A}} \quad (4.9)$$

An implementation of the solver in python compared to the linear method compared to each other is shown in Figure 12. But the K_A in this method is the reciprocal of the actual k-factor. And by setting all the function to equal zeros a non-linear root solver can be used, and Python framework has a library called SciPy's3 which has a popular optimize library that have root solving / optimizations algorithms implemented and available for the use to solve the non-linear solver.

[3]See SciPy's documentation of the method https://docs.scipy.org/doc/scipy/reference/optimize.html, but it is numerically based optimization algorithm, if this is of interest also see Newtons-Raphson method and Secant method

In [50]:
```
#Test data!
A_test = np.array([150,0,100])
B_test = np.array([100,100,0])
C_test = np.array([0.0,105.0,105])
ref_test = np.array([270.0,260.0,250.0])
```

In [51]:
```
def SolveInstance(A,B,C,ref,x0=(1.00,1.00,1.00)):
    """
    Solves an instance, A,B,C,ref all contain an 1x3 numpy array where the index is the tril index,

    x0 is a 3x tuple with initial guesses for each K-factors
    """
    alpha = np.array([(A[0]+B[0])/ref[0],(B[1]+C[1])/ref[1],(C[2]+A[2])/ref[2]])
    beta = np.array([A[0]/B[0],B[1]/C[1],C[2]/A[2]])
    gamma = np.array([ref[0]/A[0],ref[1]/B[1],ref[2]/C[2]])

    def equations(x):
        K_a,K_b,K_c = x # The unkown variables

        f_AB = 1/(1/(alpha[0]-(1/((1/K_b)+((1/K_a)*beta[0]))))-(gamma[0]-1/K_a))-K_a
        f_BC = 1/(1/(alpha[1]-(1/((1/K_c)+((1/K_b)*beta[1]))))-(gamma[1]-1/K_b))-K_b
        f_CA = 1/(1/(alpha[2]-(1/((1/K_a)+((1/K_c)*beta[2]))))-(gamma[2]-1/K_c))-K_c

        return(f_AB,f_BC,f_CA)

    Cal = optimize.fsolve(equations,x0)
    #This method calculates the inverse of the K factors, so the values get inverted
    Cal = 1/Cal
    #Returns the values
    return Cal

Cal = SolveInstance(A_test,B_test,C_test,ref_test)
print(str(Cal))
```

```
[1.04       1.14       1.39047619]
```

In [53]:
```
def LinearSolver(A,B,C,ref):
    TheMatrix = np.vstack((A,B,C)).transpose()
    TheMatrixInverse = np.linalg.inv(TheMatrix)
    K = np.dot(TheMatrixInverse, ref)
    return K

LinCal = LinearSolver(A_test,B_test,C_test,ref_test)
print(LinCal)
```

```
[1.04       1.14       1.39047619]
```

Figure 12 - Implementation of the non-linear solver, compared to the result to the linear method, and the results achieve are the same

Source Code

Source Code – Parallel calibration

Source Code – Calibration Statistics

5 Source Code – Parallel calibration

Printed 12/05-2019

```
# -*- coding: utf-8 -*-
"""
Created on for Stig Harald Gustavsens Master Thesis - 2019

@author: stig
"""
#Adding standard use libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns

from datetime import datetime
from datetime import timedelta

import os  #For file handeling and saving
import copy

from Streams import *
from DatapointsToolbox import *
from CalibrationStatistics import *



"""

------------ Parralell Calibration Methods ------------------------

"""


def find_nearest(array, value):
    """ This method uses the powers of numpy and takes an array and find and returns the index of the closes value given as a parameter """
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx

def SyncPhase(elapsed):
    """
    This method gets a list of the elapsed times for the a phase form each of the streams and
    syncronizes the elapsed time of the streams towards the stream with the smallest number of data points,
    and fills this into a matrix of values of the inedcis witch correspond to each of the syncronized timestamp
    """
    #find the lowest index.
    Smallest = np.inf
    cnt= 0
    for times in elapsed:
        if (len(times)<Smallest):
            smallest = len(times)
            SmallestStream = cnt
        cnt = cnt+1

    #Find the smallest difference between elapsedtime
    cnt2 = 0
```

```
57.         for times in elapsed:
58.             if(len(times)==smallest):
59.                 v = np.arange(0,smallest)  #Creates  vectors for the indexes
60.             else:
61.                 v = np.zeros(smallest)
62.
63.             if(cnt2>=1):
64.                 v = np.vstack((last,v))
65.
66.             last = v
67.             cnt2 = cnt2+1
68.
69.         indecies = v#.transpose()
70.
71.         Melapsed = elapsed[SmallestStream]
72.
73.         cnt3 = 0
74.         for stream in elapsed:
75.             if(len(stream) == len(Melapsed)): #Jump over the referance stream
76.                 continue
77.             else:
78.                 for i in range(len(Melapsed)):
79.                     indecies[cnt3,i] = find_nearest(stream,Melapsed[i])
80.
81.             cnt3 = cnt3+1
82.
83.         return indecies.transpose()
84.
85.
86.     def SyncTrial(Streams):
87.         """
88.         This Method goes systematically through all the Phases and get the syncronized ind
    ex values from the SyncPhase for each of the phases
89.
90.         """
91.
92.         Oilelapsed = []
93.         Gaselapsed = []
94.         Wtrelapsed = []
95.
96.         #Vectorize elapsed time
97.         for stream in Streams:
98.             Oilelapsed.append(stream.OilMass.data['elapsed'].values)
99.             Gaselapsed.append(stream.GasMass.data['elapsed'].values)
100.            Wtrelapsed.append(stream.WtrMass.data['elapsed'].values)
101.
102.        SyncdOil = SyncPhase(Oilelapsed)
103.        SyncdGas = SyncPhase(Gaselapsed)
104.        SyncdWtr = SyncPhase(Wtrelapsed)
105.
106.        return(SyncdOil,SyncdGas,SyncdWtr)
107.
108.
109.    def GetRuns(SyncedPhases, Streams):
110.        """
111.        This Uses the Syncronized phases of each of the streams and build up the tiral Mat
    rices for each of the phases
112.        And returns The a touple of Each Phase in the streams, to create the values for th
    e streams in a given paralell calibration Trial.
113.        """
114.
115.        (SyOil,SyGas,SyWtr) = SyncedPhases
116.
117.        Orun = np.zeros(SyOil.shape)
118.        for i in range(len(Orun[:,1])):
119.            j = 0
```

```
120.            for stream in Streams:
121.                Orun[i,j] = stream.OilMass.data.loc[SyOil[i,j],'cumulative']
122.                j = j+1
123.
124.        Grun = np.zeros(SyGas.shape)
125.        for i in range(len(SyGas[:,1])):
126.            j = 0
127.            for stream in Streams:
128.                Grun[i,j] = stream.GasMass.data.loc[SyGas[i,j],'cumulative']
129.                j = j+1
130.
131.        Wrun = np.zeros(SyWtr.shape)
132.        for i in range(len(SyWtr[:,1])):
133.            j = 0
134.            for stream in Streams:
135.                Wrun[i,j] = stream.WtrMass.data.loc[SyWtr[i,j],'cumulative']
136.                j = j+1
137.
138.        return (Orun,Grun,Wrun)
139.
140.    def ParralellTrial(Streams):
141.        """ This method performes a generation of a paralell trial data where the"""
142.        TrialMatrixes = GetRuns(SyncTrial(Streams),Streams)
143.        return(TrialMatrixes)
144.
145.
146.
147.    def GetElapsed(SyncedPhases, Streams):
148.        """Returns a parralell calibration elapsed times for each data point, mainly to sy
        ncronize data when plotting and correctly placing the x-axis"""
149.        (SyOil,SyGas,SyWtr) = SyncedPhases
150.
151.        Orun = np.zeros(SyOil.shape)
152.        for i in range(len(Orun[:,1])):
153.            j = 0
154.            for stream in Streams:
155.                Orun[i,j] = stream.OilMass.data.loc[SyOil[i,j],'elapsed']
156.                j = j+1
157.
158.        Grun = np.zeros(SyGas.shape)
159.        for i in range(len(SyGas[:,1])):
160.            j = 0
161.            for stream in Streams:
162.                Grun[i,j] = stream.GasMass.data.loc[SyGas[i,j],'elapsed']
163.                j = j+1
164.
165.        Wrun = np.zeros(SyWtr.shape)
166.        for i in range(len(SyWtr[:,1])):
167.            j = 0
168.            for stream in Streams:
169.                Wrun[i,j] = stream.WtrMass.data.loc[SyWtr[i,j],'elapsed']
170.                j = j+1
171.
172.        return (Orun,Grun,Wrun)
173.
174.
175.    def ParralellTrialElapsed(Streams): #Returs a tuple of elapsed times for each datapoin
        t in a parralell calibration.
176.        TrialElapsedMatrixes = GetElapsed(SyncTrial(Streams),Streams)
177.        return(TrialElapsedMatrixes)
178.
179.    """  """
180.
181.    def LinearSolver(A,B,C,ref):
182.
183.        TheMatrix = np.vstack((A,B,C))
```

```
184.        TheMatrixInverse = np.linalg.inv(TheMatrix) #May become singular and non-
    invertable?
185.        K = np.dot(TheMatrixInverse, ref)
186.        return K
187.    """
188.
189.
190.    Under her kommer det som skal pakkes inn i egen Parralell kalibrerings klasse.
191.
192.
193.    """
194.    def FindStreamNames(TrialStreams):
195.        """
196.        A small method wich goes trough a set of Trial streams and separates out all the s
    treams in each trial and collect all the names of the streams
197.        """
198.        StreamNames = []
199.        for Trial in TrialStreams:
200.            for Stream in Trial:
201.                if Stream.name not in StreamNames:
202.                    StreamNames.append(Stream.name)
203.
204.        StreamNames.remove('Third Party Separator')
205.        return StreamNames
206.
207.    def BuildMFrame(TrialStreams):
208.        """
209.        This method finds all the stream names and checks the streams inside the list of a
    lle the TrialStreams and the order to and
210.        and arranges and creates a M matrix from of zeros and ones in order to know witch
    values to be filled later.
211.        where 0 remain zero and ones will be filled with cumulative values for the solver.

212.
213.        """
214.        StreamNames = FindStreamNames(TrialStreams)
215.        first = True
216.        for Trial in TrialStreams:
217.            #Checker = StreamNames
218.            row = np.zeros(len(StreamNames))
219.
220.            for Stream in Trial:
221.                if(Stream.name == "Third Party Separator"): #Skips the refreance measurmen
    t since this measurment goes to the lowercase m vector (refreance vector)
222.                    continue
223.                StreamIndex = StreamNames.index(Stream.name)
224.                row[StreamIndex] = 1
225.
226.            if(first):
227.                M_frame = row
228.
229.            else:
230.                M_frame = np.vstack((M_frame,row))
231.            first = False
232.
233.        return M_frame
234.
235.    def GetM_m(Frame, PTrial, phase):
236.        """
237.        Creats a list of tuples of M matrix and m vectors witch can be inserted into the p
    arralell calibration solver.
238.        """
239.        Mm = []
240.        for i in range(0, min(len(trial[phase]) for trial in PTrial)):
241.            #Do the thing
242.
```

```
243.            M = Frame.astype(float)  #Copys the frame and sets values as float variables i
    nto the M matrix where the ones will be filled and the zeros will remain zero
244.            m = np.zeros(len(Frame)) #Creates a m vector for the separator accumulated val
    ues for the given trial instance.
245.            for j in range(len(Frame)):
246.
247.                cnt = 0
248.                for k in range(len(Frame[j])):
249.                    if (Frame[j,k] == 0):  #If the value is zero meas the stream has not b
    een flowing towards the separator this stream
250.                        continue
251.                    else:
252.                        M[j,k] = PTrial[j][phase][i][cnt] #Sets the give accumulated value
     for the spesific stream into the M matrix
253.                        cnt = cnt + 1 # A count whitch takes care of the correct placement
     in the M matrix as well as inn the m vector
254.                m[j] = PTrial[j][phase][i][cnt]  #Fills the referance accumulated from the
     separator into m vector
255.            Mm.append((M,m))
256.        return Mm
257.
258.
259.    def Solver(Mm): #Linear solver the parameter is a touple of M matrix and m vector
260.        """
261.        Solves the system of equation to get the kalibration values
262.        """
263.        TheMatrix, m = Mm #Tuple unpacking of matrix and vector
264.        TheMatrixInverse = np.linalg.inv(TheMatrix) #May become singular and non-
    invertable?
265.        K = np.dot(TheMatrixInverse, m)
266.        return K
267.
268.    def ParralellCalibrate(TrialStreams):
269.        """
270.        A general solver witch get a list of TrialStreams and continues on to solve parral
    ell calibration for the spesific instance.
271.        """
272.        Trials = []
273.        for TrialStream in TrialStreams:
274.            Trials.append(ParrallellTrial(TrialStream)) #Syncronicing timestamps and locati
    ng data.
275.
276.        Frame = BuildMFrame(TrialStreams) #Buids the M frame for the
277.
278.        Kfactors = []
279.        for phase in range(3): # 3 phases Oil, Gas and Water in that order.
280.            Mm = GetM_m(Frame,Trials,phase) #Gets a touple of the M matrix and m phase
281.            first = True
282.            for i in range(1,len(Mm)):
283.                try:
284.                    K_i = Solver(Mm[i])
285.                except:
286.                    print("Singular matrix at "+str(i)+"iteration")
287.                    continue
288.
289.                if(first):
290.                    K = K_i
291.                else:
292.                    K = np.vstack((K,K_i))
293.                first = False
294.
295.            Kfactors.append(K)
296.        return Kfactors
297.
298.    """
299.
```

```
300.    Ferdig med parralell kalibrering metoden
301.
302.
303.    """
304.    class ParralellCalibration:
305.        """
306.
307.        This is a Class to create an istance of a Calibration object, in order to simplify
         the reporting and documentation of the method as well as further analysis of data.
308.
309.        """
310.        def __init__(self,TrialStreams):
311.
312.            #Static Values
313.            self.phases = ["Oil","Gas","Water"]
314.            self.Colors = ['#000000','#DF0071','#00A030','#EE7900','#777777',' #000028','#
        B9B9B9','#E6E6E6']
315.            self.DateCalculated = datetime.now()
316.
317.            #Perform Parralell Calibration
318.            self.StreamNames = FindStreamNames(TrialStreams)
319.            self.TrialStreams = TrialStreams
320.
321.            #
322.            # Add order Streams in trial funksjon here! so that the streams come in the co
        rrect order for the algorithm
323.            #
324.
325.            self.Elapsed = []
326.            for TrialStream in TrialStreams:
327.                self.Elapsed.append(ParralellTrialElapsed(TrialStream)) #Getting timestamp
        s of Elapsed time
328.            self.Kfaktors = ParralellCalibrate(TrialStreams)
329.            self.frame = BuildMFrame(TrialStreams) #Buids the M frame for the
330.
331.            #Storeing the final k-factor - just the last instance.
332.            self.lastKfactors = []
333.            for i in range(len(self.StreamNames)):
334.                self.lastKfactors.append(np.array([self.Kfaktors[0][len(self.Kfaktors[0])-
        1,i],self.Kfaktors[1][len(self.Kfaktors[1])-
        1,i],self.Kfaktors[2][len(self.Kfaktors[2])-1,i]]))
335.
336.
337.            #Parralell calibration excecuted.
338.
339.        def Plot(self,save=False):
340.            for i in range(len(self.StreamNames)):
341.                plt.plot(self.Kfaktors[0][50:,i], label ='Oil', color='#DF0071')
342.                plt.plot(self.Kfaktors[1][50:,i], label = 'Gas', color='#00A030')
343.                plt.plot(self.Kfaktors[2][50:,i], label ='Water',color='#EE7900')
344.                plt.xlabel('Runs')
345.                plt.ylabel('K-factor')
346.                plt.title(self.StreamNames[i]+' calibration')
347.                plt.legend()
348.                plt.grid()
349.                plt.show()
350.                plt.clf()#Clares all the axis in the figure for a new iteration of plottin
        g
351.                if(save):
352.                    plt.savefig(self.StreamNames[i]+"_Kfactors.png")
353.                plt.show()
354.
355.
356.        def PlotStreams(self,save=False,figuresize=(15,10)):
357.            """
```

```
358.            This Method Plots the Flowrates in the matrix for for each of the trial includ
    ing the separator
359.            """
360.            for phase in range(len(self.phases)): #For Each Phase
361.                SquareMatrixSize = len(self.Elapsed)
362.                fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figures
    ize)
363.
364.                fig.suptitle(self.phases[phase]+" Parralell calibration mass flow rate dat
    a",fontsize=15)
365.                for i in range(SquareMatrixSize):  # For each Trial
366.                    cnt = 0
367.                    for j in range(SquareMatrixSize+1): # For each column in axis matrix

368.                        try:
369.                            #Plotting the Separator values
370.                            if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
371.                                if(i==0): #Sets the name of the stream at the top.
372.                                    ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][
    cnt].name)
373.
374.                                #Checks the phase and plots accordingly
375.                                if(self.phases[phase]=="Oil" and self.TrialStreams[i][cnt]
    .name == 'Third Party Separator'):
376.                                    ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    OilMass.data['value'],color=self.Colors[phase])
377.
378.                                if(self.phases[phase]=="Gas" and self.TrialStreams[i][cnt]
    .name == 'Third Party Separator'):
379.                                    ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    GasMass.data['value'],color=self.Colors[phase])
380.
381.                                if(self.phases[phase]=="Water"and self.TrialStreams[i][cnt
    ].name == 'Third Party Separator'):
382.                                    ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    WtrMass.data['value'],color=self.Colors[phase])
383.
384.                                cnt= cnt+1
385.                                continue
386.
387.
388.
389.                            if(j<SquareMatrixSize): #If within the MPFM Streams and not in
     the separators
390.                                if(i==0):
391.                                    ax[0,j].set_title(self.StreamNames[j])
392.                                if(self.frame[i,j]==0): #If it is Zero then no need to plo
    t, and jumt to next axsis with incrementing the trial, because the trial has no been p
    lotted
393.                                    ax[i,j].set_xticklabels([]) #removing the x ticks on x
    axis
394.                                    ax[i,j].set_yticklabels([]) #removing the y ticks on y
    axis
395.                                    continue
396.                                else:
397.                                    TheStreamIndex = self.StreamNames.index(self.TrialStre
    ams[i][cnt].name)
398.                                    #TrialStreams[i][j].OilMass.data['cumulative']
399.                                    if(self.phases[phase]=="Oil"):
400.                                        ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
    ].OilMass.data['value'],color=self.Colors[phase])
401.                                    if(self.phases[phase]=="Gas"):
402.                                        ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
    ].GasMass.data['value'],color=self.Colors[phase])
403.                                    if(self.phases[phase]=="Water"):
```

```
404.                                     ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
    ].WtrMass.data['value'],color=self.Colors[phase])
405.                             cnt = cnt+1
406.
407.                         except IndexError:
408.                             cnt = cnt+1
409.                             continue
410.                 if(save):
411.                     fig.savefig(title+".png")
412.                 fig.show()
413.
414.         def PlotCumulativeMatrix(self,save=False,figuresize=(12,8)):
415.             """
416.             This Method Plots the Flowrates in the matrix for for each of the trial includ
    ing the separator
417.             """
418.             for phase in range(len(self.phases)): #For Each Phase
419.                 SquareMatrixSize = len(self.Elapsed)
420.                 fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figures
    ize)
421.
422.                 fig.suptitle(self.phases[phase]+" Parralell calibration mass flow rate dat
    a",fontsize=15)
423.                 for i in range(SquareMatrixSize): #For each Trial
424.                     cnt = 0
425.                     for j in range(SquareMatrixSize+1): #For each Stream in Trial / column
     in axis matrix
426.                         try:
427.                             #Plotting the Separator values
428.                             if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
429.                                 if(i==0): #Sets the name of the stream at the top.
430.                                     ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][
    cnt].name)
431.
432.                                 #Checks the phase and plots accordingly
433.                                 if(self.phases[phase]=="Oil" and self.TrialStreams[i][cnt]
    .name == 'Third Party Separator'):
434.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    OilMass.data['cumulative'],color=self.Colors[phase])
435.
436.                                 if(self.phases[phase]=="Gas" and self.TrialStreams[i][cnt]
    .name == 'Third Party Separator'):
437.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    GasMass.data['cumulative'],color=self.Colors[phase])
438.
439.                                 if(self.phases[phase]=="Water"and self.TrialStreams[i][cnt
    ].name == 'Third Party Separator'):
440.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
    WtrMass.data['cumulative'],color=self.Colors[phase])
441.
442.                                 cnt= cnt+1
443.                                 continue
444.
445.
446.                             if(j<SquareMatrixSize): #If within the MPFM Streams and not in
     the separators
447.                                 if(i==0):
448.                                     ax[i,j].set_title(self.StreamNames[j])
449.                                 if(self.frame[i,j]==0): #If it is Zero then no need to plo
    t, and jumt to next axsis with incrementing the trial, because the trial has no been p
    lotted
450.                                     ax[i,j].set_xticklabels([]) #removing the x ticks on x
    axis
451.                                     ax[i,j].set_yticklabels([]) #removing the y ticks on y
    axis
452.                                     continue
```

```
453.                                 else:
454.                                     TheStreamIndex = self.StreamNames.index(self.TrialStre
       ams[i][cnt].name)
455.                                     #TrialStreams[i][j].OilMass.data['cumulative']
456.                                     if(self.phases[phase]=="Oil"):
457.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
       ].OilMass.data['cumulative'],color=self.Colors[phase])
458.                                     if(self.phases[phase]=="Gas"):
459.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
       ].GasMass.data['cumulative'],color=self.Colors[phase])
460.                                     if(self.phases[phase]=="Water"):
461.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
       ].WtrMass.data['cumulative'],color=self.Colors[phase])
462.                                 cnt = cnt+1
463.
464.                             except IndexError:
465.                                 cnt = cnt+1
466.                                 continue
467.                     #fig.tight_layout()
468.                     fig.show()
469.
470.
471.
472.         def PlotOneCumulativeMatrix(self,save=False,figuresize=(12,8)):
473.             """
474.             This Method Plots the Flowrates in the matrix for for each of the trial includ
       ing the separator
475.             """
476.             titlenames = FindStreamNames(self.TrialStreams)
477.             legend = True
478.             SquareMatrixSize = len(self.Elapsed)
479.             fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figuresize)
480.
481.             fig.suptitle(" Parralell calibration cumulative mass [KiloTonnes]",fontsize=15
       )
482.             for i in range(SquareMatrixSize): #For each Trial
483.                 cnt = 0
484.                 for j in range(SquareMatrixSize+1): #For each Stream in Trial / column in
       axis matrix
485.                     try:
486.                         #Plotting the Separator values
487.                         if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
488.                             if(i==0): #Sets the name of the stream at the top.
489.                                 ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][cnt]
       .name)
490.
491.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].OilMass.
       data['cumulative']/1000/1000,color=self.Colors[0])
492.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].GasMass.
       data['cumulative']/1000/1000,color=self.Colors[1])
493.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].WtrMass.
       data['cumulative']/1000/1000,color=self.Colors[2])
494.                             ax[i,j].set_xticklabels([])
495.
496.                             cnt= cnt+1
497.                             continue
498.
499.
500.                         if(j<SquareMatrixSize): #If within the MPFM Streams and not in the
        separators
501.                             if(self.StreamNames[j] in titlenames and len(titlenames) != 0)
       :
502.                                 ax[0,j].set_title(self.StreamNames[j])
503.                                 titlenames.remove(self.StreamNames[j])
504.
```

```
505.                             if(self.frame[i,j]==0): #If it is Zero then no need to plot, a
    nd jumt to next axsis with incrementing the trial, because the trial has no been plott
    ed
506.                                 if(legend):
507.                                     ax[i,j].plot([],[],label='Oil',color=self.Colors[0])
508.                                     ax[i,j].plot([],[],label='Gas',color=self.Colors[1])
509.                                     ax[i,j].plot([],[],label='Water',color=self.Colors[2])

510.                                     ax[i,j].legend(fontsize = 'x-large')
511.                                     legend = False
512.                                 ax[i,j].set_xticklabels([]) #removing the x ticks on xaxis

513.                                 ax[i,j].set_yticklabels([]) #removing the y ticks on yaxis

514.                                 continue
515.                             else:
516.                                 TheStreamIndex = self.StreamNames.index(self.TrialStreams[
    i][cnt].name)
517.                                 #TrialStreams[i][j].OilMass.data['cumulative']
518.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].OilMas
    s.data['cumulative']/1000/1000,color=self.Colors[0])
519.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].GasMas
    s.data['cumulative']/1000/1000,color=self.Colors[1])
520.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].WtrMas
    s.data['cumulative']/1000/1000,color=self.Colors[2])
521.                                 ax[i,j].set_xticklabels([])
522.                         cnt = cnt+1
523.
524.                     except IndexError:
525.                         cnt = cnt+1
526.                         continue
527.             if(save):
528.                 fig.savefig(title+".png")
529.             fig.show()
530.
531.         def PlotOneStreamMatrix(self,save=False,figuresize=(12,8)):
532.             """
533.             This Method Plots the Flowrates in the matrix for for each of the trial includ
    ing the separator
534.             """
535.             titlenames = FindStreamNames(self.TrialStreams)
536.             legend = True
537.             SquareMatrixSize = len(self.Elapsed)
538.             fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figuresize)

539.
540.             fig.suptitle(" Parralell calibration mass flow rate [Tonnes/hour]",fontsize=15
    )
541.             for i in range(SquareMatrixSize): #For each Trial
542.                 cnt = 0
543.                 for j in range(SquareMatrixSize+1): #For each Stream in Trial / column in
    axis matrix
544.                     try:
545.                         #Plotting the Separator values
546.                         if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
547.                             if(i==0): #Sets the name of the stream at the top.
548.                                 ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][cnt]
    .name)
549.
550.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].OilMass.
    data['timestamp'],self.TrialStreams[i][cnt].OilMass.data['value']/1000,color=self.Colo
    rs[0],zorder=1)
551.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].GasMass.
    data['timestamp'],self.TrialStreams[i][cnt].GasMass.data['value']/1000,color=self.Colo
    rs[1],zorder=2)
```

```
552.                             ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].WtrMass.
      data['timestamp'],self.TrialStreams[i][cnt].WtrMass.data['value']/1000,color=self.Colo
      rs[2],zorder=0)
553.                             ax[i,j].set_xticklabels([])
554.
555.                             cnt= cnt+1
556.                             continue
557.
558.
559.                         if(j<SquareMatrixSize): #If within the MPFM Streams and not in the
       separators
560.                             if(self.StreamNames[j] in titlenames and len(titlenames) != 0)
      :
561.                                 ax[0,j].set_title(self.StreamNames[j])
562.                                 titlenames.remove(self.StreamNames[j])
563.                             if(self.frame[i,j]==0): #If it is Zero then no need to plot, a
      nd jumt to next axsis with incrementing the trial, because the trial has no been plott
      ed
564.                                 if(legend):
565.                                     ax[i,j].plot([],[],label='Oil',color=self.Colors[0])
566.                                     ax[i,j].plot([],[],label='Gas',color=self.Colors[1])
567.                                     ax[i,j].plot([],[],label='Water',color=self.Colors[2])

568.                                     ax[i,j].legend(fontsize = 'x-large')
569.                                     legend = False
570.                                 ax[i,j].set_xticklabels([]) #removing the x ticks on xaxis

571.                                 ax[i,j].set_yticklabels([]) #removing the y ticks on yaxis

572.                                 continue
573.                             else:
574.                                 TheStreamIndex = self.StreamNames.index(self.TrialStreams[
      i][cnt].name)
575.                                 #TrialStreams[i][j].OilMass.data['cumulative']
576.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].OilMas
      s.data['timestamp'],self.TrialStreams[i][cnt].OilMass.data['value']/1000,color=self.Co
      lors[0],zorder=1)
577.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].GasMas
      s.data['timestamp'],self.TrialStreams[i][cnt].GasMass.data['value']/1000,color=self.Co
      lors[1],zorder=2)
578.                                 ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].WtrMas
      s.data['timestamp'],self.TrialStreams[i][cnt].WtrMass.data['value']/1000,color=self.Co
      lors[2],zorder=0)
579.                                 ax[i,j].set_xticklabels([])
580.                         cnt = cnt+1
581.
582.                     except IndexError:
583.                         cnt = cnt+1
584.                         continue
585.             if(save):
586.                 fig.savefig(title+".png")
587.             fig.show()
588.
589.         def PlotMatrix(self,save=False,figuresize=(15,10)):
590.             """
591.             This Method Plots the Cumulative Matrices used in the method
592.             """
593.             titlenames = FindStreamNames(self.TrialStreams)
594.             for phase in range(len(self.phases)): #For Each Phase
595.                 SquareMatrixSize = len(self.Elapsed)
596.                 fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize,figsize=figuresiz
      e)
597.
598.                 fig.suptitle(self.phases[phase]+" Parralell calibration trial matrix data"
      ,fontsize=15)
599.                 for i in range(SquareMatrixSize): #For each Trial
```

```
600.                     cnt=0
601.                     for j in range(SquareMatrixSize): #For each Stream in Trial
602.                         if(self.StreamNames[j] in titlenames and len(titlenames) != 0):
603.                             ax[0,j].set_title(self.StreamNames[j])
604.                             titlenames.remove(self.StreamNames[j])
605.                         if(self.frame[i,j]==0):
606.                             ax[i,j].set_xticklabels([]) #removing the x ticks on xaxis
607.                             ax[i,j].set_yticklabels([]) #removing the y ticks on yaxis
608.                             continue
609.                         else:
610.                             if(self.TrialStreams[i][cnt].name not in self.StreamNames):
611.                                 continue #it is a separator stream.
612.                             else:
613.                                 TheStreamIndex = self.StreamNames.index(self.TrialStreams[
      i][cnt].name)
614.                                 #TrialStreams[i][j].OilMass.data['cumulative']
615.                                 if(self.phases[phase]=="Oil"):
616.                                     ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].Oi
      lMass.data['cumulative'],color=self.Colors[phase])
617.                                 if(self.phases[phase]=="Gas"):
618.                                     ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].Ga
      sMass.data['cumulative'],color=self.Colors[phase])
619.                                 if(self.phases[phase]=="Water"):
620.                                     ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt].Wt
      rMass.data['cumulative'],color=self.Colors[phase])
621.                                 cnt = cnt +1
622.                 if(save):
623.                     fig.savefig(title+".png")
624.                 fig.show()
625.
626.         def PlotStreamsIntVar(self,save=False,figuresize=(15,10)):
627.             """
628.             This Method Plots the Flowrates in the matrix for for each of the trial includ
      ing the separator
629.             """
630.             titlenames = FindStreamNames(self.TrialStreams)
631.             legend = True
632.             for phase in range(1):  #Only need one output for entire calibration data. it
      is the -
633.                 SquareMatrixSize = len(self.Elapsed)
634.                 fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figures
      ize)
635.                 #fig.autofmt_xdate()
636.
637.                 fig.suptitle("Parralell calibration process conditions",fontsize=15)
638.                 for i in range(SquareMatrixSize): #For each Trial
639.                     cnt = 0
640.                     for j in range(SquareMatrixSize+1): #For each Stream in Trial
641.                         try:
642.                             #Plotting the Separator values
643.                             if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
644.                                 if(i==0): #Sets the name of the stream at the top.
645.                                     ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][
      cnt].name)
646.
647.                                 #Checks the phase and plots accordingly
648.                                 if(self.phases[phase]=="Oil" and self.TrialStreams[i][cnt]
      .name == 'Third Party Separator'):
649.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      wc['timestamp'],self.TrialStreams[i][cnt].wc['value'],label ='Water cut[Vol%]',color=s
      elf.Colors[0])
650.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorOilLevel['timestamp'],self.TrialStreams[i][cnt].SeparatorOilLevel['value'],la
      bel ='Oil Level[%]',color=self.Colors[1])
```

```
651.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorWaterLevel['timestamp'],self.TrialStreams[i][cnt].SeparatorWaterLevel['value'
      ],label ='Water Level[%]',color=self.Colors[2])
652.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorPressure['timestamp'],self.TrialStreams[i][cnt].SeparatorPressure['value'],la
      bel ='Pressure [Barg]',color=self.Colors[3])
653.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorTemp['timestamp'],self.TrialStreams[i][cnt].SeparatorTemp['value'],label ='Te
      mperatur [°C]',color=self.Colors[4])
654.                                     #ax[i,SquareMatrixSize].legend()
655.
656.
657.                                 if(self.phases[phase]=="Gas" and self.TrialStreams[i][cnt]
      .name == 'Third Party Separator'):
658.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorPressure['timestamp'],self.TrialStreams[i][cnt].SeparatorPressure['value'],la
      bel ='Pressure [Barg]',color=self.Colors[0])
659.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorTemp['timestamp'],self.TrialStreams[i][cnt].SeparatorTemp['value'],label ='Te
      mperatur [°C]',color=self.Colors[1])
660.                                     ax[i,SquareMatrixSize].legend()
661.
662.
663.                                 if(self.phases[phase]=="Water"and self.TrialStreams[i][cnt
      ].name == 'Third Party Separator'):
664.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      wc['timestamp'],self.TrialStreams[i][cnt].wc['value'],label ='Water cut[Vol%]',color=s
      elf.Colors[0])
665.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorOilLevel['timestamp'],self.TrialStreams[i][cnt].SeparatorOilLevel['value'],la
      bel ='Oil Level[%]',color=self.Colors[1])
666.                                     ax[i,SquareMatrixSize].plot(self.TrialStreams[i][cnt].
      SeparatorWaterLevel['timestamp'],self.TrialStreams[i][cnt].SeparatorWaterLevel['value'
      ],label ='Water Level[%]',color=self.Colors[2])
667.                                     #ax[i,SquareMatrixSize].legend()
668.
669.                                 ax[i,SquareMatrixSize].set_xticklabels([]) #Removes the va
      lues on the x-scale
670.                                 #ax[i,SquareMatrixSize].fmt_xdata = mdates.DateFormatter('
      %d-%h')
671.                                 cnt= cnt+1
672.                                 continue
673.
674.                         #self.SeparatorWaterLevel
675.
676.                             if(j<SquareMatrixSize): #If within the MPFM Streams and not in
       the separators
677.                                 if(self.StreamNames[j] in titlenames and len(titlenames) !
      = 0):
678.                                     ax[0,j].set_title(self.StreamNames[j])
679.                                     titlenames.remove(self.StreamNames[j])
680.                                 if(self.frame[i,j]==0): #If it is Zero then no need to plo
      t, and jumt to next axsis with incrementing the trial, because the trial has no been p
      lotted
681.
682.                                     if(legend): #Plotting the legend in the first empty ma
      trix cell
683.                                         ax[i,j].plot([],[],label ='Water cut[Vol%]',color=
      self.Colors[0])
684.                                         ax[i,j].plot([],[],label ='Oil Level[%]',color=sel
      f.Colors[1])
685.                                         ax[i,j].plot([],[],label ='Water Level[%]',color=s
      elf.Colors[2])
686.                                         ax[i,j].plot([],[],label ='Pressure [Barg]',color=
      self.Colors[3])
```

```
687.                                         ax[i,j].plot([],[],label ='Temperatur [°C]',color=
      self.Colors[4])
688.                                         ax[i,j].legend(fontsize = 'x-large')
689.                                         #ax[i,j].plot([],[],label ='Pressure [Barg]',color
      =self.Colors[3])
690.                                         legend = False
691.
692.
693.                                     ax[i,j].set_xticklabels([]) #removing the x ticks on x
      axis
694.                                     ax[i,j].set_yticklabels([]) #removing the y ticks on y
      axis
695.                                     continue
696.                                 else:
697.                                     TheStreamIndex = self.StreamNames.index(self.TrialStre
      ams[i][cnt].name)
698.                                     #TrialStreams[i][j].OilMass.data['cumulative']
699.                                     if(self.phases[phase]=="Oil"):
700.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Pressure['timestamp'],self.TrialStreams[i][cnt].Pressure['value'],label ='Pressure [
      Bar]',color=self.Colors[3])
701.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Temp['timestamp'],self.TrialStreams[i][cnt].Temp['value'],label ='Temperatur [°C]',c
      olor=self.Colors[4])
702.                                         #ax[i,TheStreamIndex].legend()
703.                                     if(self.phases[phase]=="Gas"):
704.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Pressure['timestamp'],self.TrialStreams[i][cnt].Pressure['value'],label ='Pressure [
      Bar]',color=self.Colors[3])
705.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Temp['timestamp'],self.TrialStreams[i][cnt].Temp['value'],label ='Temperatur [°C]',c
      olor=self.Colors[1])
706.                                         #ax[i,TheStreamIndex].legend()
707.                                     if(self.phases[phase]=="Water"):
708.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Pressure['timestamp'],self.TrialStreams[i][cnt].Pressure['value'],label ='Pressure [
      Bar]',color=self.Colors[3])
709.                                         ax[i,TheStreamIndex].plot(self.TrialStreams[i][cnt
      ].Temp['timestamp'],self.TrialStreams[i][cnt].Temp['value'],label ='Temperatur [°C]',c
      olor=self.Colors[1])
710.                                         #ax[i,TheStreamIndex].legend()
711.                                     ax[i,TheStreamIndex].set_xticklabels([]) #Removes the
      values on the x-scale
712.                                     #ax[i,TheStreamIndex].fmt_xdata = mdates.DateFormatter
      ('%d-%h')
713.
714.                             cnt = cnt+1
715.
716.                         except IndexError:
717.                             cnt = cnt+1
718.                             continue
719.                 if(save):
720.                     fig.savefig(title+".png")
721.                 fig.show()
722.
723.         def PlotAccHist(self,n_bins=20,save=False,figuresize=(12,8)):
724.             """
725.             This Method Plots the Flowrates in the matrix for for each of the trial includ
      ing the separator
726.             """
727.             titlenames = FindStreamNames(self.TrialStreams)
728.             for phase in range(len(self.phases)): #For Each Phase
729.                 SquareMatrixSize = len(self.Elapsed)
730.                 fig, ax = plt.subplots(SquareMatrixSize,SquareMatrixSize+1,figsize=figures
      ize)
731.
```

```
732.                 fig.suptitle(self.phases[phase]+" Parralell calibration accumulator time d
      eltas",fontsize=15)
733.                 for i in range(SquareMatrixSize): #For each Trial
734.                     cnt = 0
735.                     for j in range(SquareMatrixSize+1): #For each Stream in Trial / column
       in axis matrix
736.                         try:
737.                             #Plotting the Separator values
738.                             if(type(self.TrialStreams[i][cnt]) == SeparatorStreams):
739.                                 if(i==0): #Sets the name of the stream at the top.
740.                                     ax[i,SquareMatrixSize].set_title(self.TrialStreams[i][
      cnt].name)
741.
742.                                 #Checks the phase and plots accordingly
743.                                 if(self.phases[phase]=="Oil" and self.TrialStreams[i][cnt]
      .name == 'Third Party Separator'):
744.                                     ax[i,SquareMatrixSize].hist(self.TrialStreams[i][cnt].
      OilMass.data['dt'],color=self.Colors[phase], bins=n_bins)
745.                                     Max = max(self.TrialStreams[i][cnt].OilMass.data['dt']
      )
746.                                     #ax[i,SquareMatrixSize].text('Highest time delta:'+str
      (Max))
747.
748.                                 if(self.phases[phase]=="Gas" and self.TrialStreams[i][cnt]
      .name == 'Third Party Separator'):
749.                                     ax[i,SquareMatrixSize].hist(self.TrialStreams[i][cnt].
      GasMass.data['dt'],color=self.Colors[phase], bins=n_bins)
750.                                     Max = max(self.TrialStreams[i][cnt].GasMass.data['dt']
      )
751.                                     #ax[i,SquareMatrixSize].text('Highest time delta:'+str
      (Max))
752.
753.                                 if(self.phases[phase]=="Water"and self.TrialStreams[i][cnt
      ].name == 'Third Party Separator'):
754.                                     ax[i,SquareMatrixSize].hist(self.TrialStreams[i][cnt].
      WtrMass.data['dt'],color=self.Colors[phase], bins=n_bins)
755.                                     Max = max(self.TrialStreams[i][cnt].WtrMass.data['dt']
      )
756.                                     #ax[i,SquareMatrixSize].text('Highest time delta:'+str
      (Max))
757.
758.                                 ax[i,SquareMatrixSize].set_yscale('log')
759.                                 cnt= cnt+1
760.                                 continue
761.
762.
763.                             if(j<SquareMatrixSize): #If within the MPFM Streams and not in
       the separators
764.                                 if(self.StreamNames[j] in titlenames and len(titlenames) !
      = 0):
765.                                     ax[0,j].set_title(self.StreamNames[j])
766.                                     titlenames.remove(self.StreamNames[j])
767.                                 if(self.frame[i,j]==0): #If it is Zero then no need to plo
      t, and jumt to next axsis with incrementing the trial, because the trial has no been p
      lotted
768.                                     ax[i,j].set_xticklabels([]) #removing the x ticks on x
      axis
769.                                     ax[i,j].set_yticklabels([]) #removing the y ticks on y
      axis
770.                                     continue
771.                                 else:
772.                                     TheStreamIndex = self.StreamNames.index(self.TrialStre
      ams[i][cnt].name)
773.                                     #TrialStreams[i][j].OilMass.data['cumulative']
774.                                     if(self.phases[phase]=="Oil"):
```

```
775.                                     ax[i,TheStreamIndex].hist(self.TrialStreams[i][cnt
    ].OilMass.data['dt'],color=self.Colors[phase], bins=n_bins)
776.                                     Max = max(self.TrialStreams[i][cnt].OilMass.data['
    dt'])
777.                                     #ax[i,i,TheStreamIndex].text('Highest time delta:'
    +str(Max))
778.                                 if(self.phases[phase]=="Gas"):
779.                                     ax[i,TheStreamIndex].hist(self.TrialStreams[i][cnt
    ].GasMass.data['dt'],color=self.Colors[phase], bins=n_bins)
780.                                     Max = max(self.TrialStreams[i][cnt].GasMass.data['
    dt'])
781.                                     #ax[i,i,TheStreamIndex].text('Highest time delta:'
    +str(Max))
782.                                 if(self.phases[phase]=="Water"):
783.                                     ax[i,TheStreamIndex].hist(self.TrialStreams[i][cnt
    ].WtrMass.data['dt'],color=self.Colors[phase], bins=n_bins)
784.                                     Max = max(self.TrialStreams[i][cnt].WtrMass.data['
    dt'])
785.                                     #ax[i,i,TheStreamIndex].text('Highest time delta:'
    +str(Max))
786.                                 ax[i,TheStreamIndex].set_yscale('log')
787.                             cnt = cnt+1
788.
789.                         except IndexError:
790.                             cnt = cnt+1
791.                             continue
792.
793.                 if(save):
794.                     fig.savefig(title+".png")
795.                 fig.show()
796.
797.         def GetKfactors(self,Stream,Phase):
798.             if(Stream in self.StreamNames):
799.                 streamIndex = self.StreamNames.index(Stream)
800.             else:
801.                 print('Stream name not found')
802.                 return 'Stream name not found'
803.             if(Phase in self.phases):
804.                 PhaseIndex = self.phases.index(Phase)
805.             else:
806.                 print('Phase not found')
807.                 return 'phase not found'
808.
809.             return self.Kfaktors[PhaseIndex][:,streamIndex]
810.
811.
812.         def PrintMethodReport(self):
813.             """
814.             This print out a report of the parralell calibration
815.
816.             """
817.             for i in range(len(self.StreamNames)):
818.                 print("--------------------------"+self.StreamNames[i]+"----------------
    -------")
819.                 print(" Oil K-factor = "+str(self.Kfaktors[0][len(self.Kfaktors[0])-
    1,i]))
820.                 print(" Gas K-factor = "+str(self.Kfaktors[1][len(self.Kfaktors[1])-
    1,i]))
821.                 print(" Water K-factor = "+str(self.Kfaktors[2][len(self.Kfaktors[2])-
    1,i]))
822.                 print("------------------------------------------------------------")
823.
824.         def Statisics(self):
825.
826.             self.StatStreams = []
827.             for i in range(len(self.StreamNames)):
```

```
828.                 Stream = []
829.                 Stream.append(stats(self.Kfaktors[0][0:,i],self.StreamNames[i], self.phase
      s[0]))
830.                 Stream.append(stats(self.Kfaktors[1][0:,i],self.StreamNames[i], self.phase
      s[1]))
831.                 Stream.append(stats(self.Kfaktors[2][0:,i],self.StreamNames[i], self.phase
      s[2]))
832.                 self.StatStreams.append(Stream)
833.
834.             self.PlotStats()
835.             self.PrintStats()
836.             self.CreateStatsDF()
837.
838.         def PlotStats(self):
839.
840.             for stream in self.StatStreams:
841.                 for phase in stream:
842.                     phase.plot()
843.
844.         def PrintStats(self):
845.
846.             for stream in self.StatStreams:
847.                 for phase in stream:
848.                     print(phase.to_string())
849.
850.         def GetStats(self,Stream,Phase):
851.             if(Stream in self.StreamNames):
852.                 streamIndex = self.StreamNames.index(Stream)
853.             else:
854.                 print('Stream name not found')
855.                 return 'Stream name not found'
856.             if(Phase in self.phases):
857.                 PhaseIndex = self.phases.index(Phase)
858.             else:
859.                 print('Phase not found')
860.                 return 'Phase not found'
861.
862.             return self.StatStreams[streamIndex][PhaseIndex]
863.
864.         def CreateStatsDF(self):
865.             """ Creates a resulting multi dimentional Pandas datafram of the result from t
      he statistics """
866.             StreamNames = []
867.             Phase = []
868.             first = True
869.             for Stream in self.StatStreams:
870.                 for result in Stream:
871.                     StreamNames.append(result.name)
872.                     Phase.append(result.phase)
873.                     if(first):
874.                         ResultMatrix = result.ResultVector()
875.                         first = False
876.                     else:
877.                         ResultMatrix = np.vstack((ResultMatrix,result.ResultVector()))
878.
879.             MdimIndecis = list(zip(StreamNames,Phase))
880.             MdimIndecis = pd.MultiIndex.from_tuples(MdimIndecis)
881.             self.StatsDF = pd.DataFrame(ResultMatrix,MdimIndecis,['mean','uncert','St.dev'
      ,'SampleSize'])
882.             self.StatsDF.index.names = ['Stream','Phase']
883.
884.             # self.StatsDF.xs('Oil',level='Phase')  returnere alle olje kfaktor linjene
885.             # self.StatsDF.loc('Bøyla') returnere en normal DF med bøyla talla
886.
887.         def CreateBasisDF(self):
888.
```

```
889.            dfs = []
890.            for Stream in self.StatStreams:
891.                for phase in Stream:
892.                    thisBase = phase.Basis
893.                    stream = []
894.                    streamphase = []
895.                    for i in range(len(thisBase)):
896.                        stream.append(phase.name)
897.                        streamphase.append(phase.phase)
898.                        data = {'stream':stream, 'phase':streamphase, 'k-
    factor':thisBase.tolist()}
899.                    dfs.append(pd.DataFrame(data))
900.
901.            self.BasisDF = pd.concat(dfs, ignore_index=True)
902.
903.
904.        def ViolinPlot(self,FigureSize = (10,10)):
905.            self.CreateBasisDF()
906.            plt.figure(figsize=FigureSize)
907.            ax = sns.violinplot(x='stream', y='k-
    factor', hue ='phase', data=self.BasisDF)
908.            plt.show()
909.
910.        def Stats(self):
911.            self.PlotStats()
912.            self.PrintStats()
913.
914.        def Report(self,title = 'Parrell calibration report'):
915.            #not yet implemented
916.            print('not yet implemented')
917.
918.        def Save(self,CalibrationName):
919.            self.rootDir = os.getcwd()
920.            #self.dirname = self.rootDir+"/"+CalibrationName"_"+str(self.DateCalculated)
921.            #os.mkdir(CalibrationName"_"+str(self.DateCalculated))
922.            os.chdir(self.dirname)
923.
924.        def SavePlots(self):
925.            "Not yet implemented"
926.            return "Not yet implemented"
927.        def SaveTrials(self):
928.            "Not yet implemented"
929.            return "Not yet implemented"
930.
931.
932.
933.
934.
935.    """Calibration Comparrison methods """
936.
937.    def PlotCalvsCal(Cals,StreamName):
938.        fig, axes = plt.subplots(figsize=(15,10),dpi=300,nrows=3,ncols=1)
939.
940.        #or current_ax in axes:
941.        #    current_ax.plot(Kfaktorer[0][0:,0], label ='Oil_P', color='#DF0071')
942.        #    current_ax.plot(BøTO[1][0:],  label = 'Oil_T', color='#00A030')
943.        Cal2 = Cals[0]
944.        Cal1 = Cals[1]
945.
946.        axes[0].plot(Cal1.GetKfactors(StreamName,'Oil')[20:], label ='Oil_Trad', color='#D
    F0071')
947.        axes[0].plot(Cal2.GetKfactors(StreamName,'Oil')[20:], label ='Oil_Para', color='#0
    0A030')
948.        #axes[0].plot(TBø[0][1][0:],  label = 'Oil_T', color='#00A030')
949.        axes[0].legend()
950.        axes[0].set_title(StreamName+' Comparrison  Calibration')
```

```
951.
952.        axes[1].plot(Cal1.GetKfactors(StreamName,'Gas')[20:], label ='Gas_Trad', color='#D
    F0071')
953.        axes[1].plot(Cal2.GetKfactors(StreamName,'Gas')[20:], label ='Gas_Para', color='#0
    0A030')
954.        #axes[1].plot(TBø[1][1][0:],  label = 'Gas_T', color='#00A030')
955.        axes[1].legend()
956.
957.        axes[2].plot(Cal1.GetKfactors(StreamName,'Water')[20:], label ='Water_Trad', color
    ='#DF0071')
958.        axes[2].plot(Cal2.GetKfactors(StreamName,'Water')[20:], label ='Water_Para', color
    ='#00A030')
959.        #axes[2].plot(TBø[2][1][0:],  label = 'Water_T', color='#00A030')
960.        axes[2].legend(fontsize = 'x-large')
961.
962.        fig.show()
963.
964.    def PlotOneCalvsCal(Cals,StreamName):
965.        plt.figure(figsize=(15,10))
966.
967.        Colors = ['#000000','#DF0071','#00A030','#EE7900','#777777',' #000028','#B9B9B9','
    #E6E6E6']
968.
969.        #or current_ax in axes:
970.        #    current_ax.plot(Kfaktorer[0][0:,0], label ='Oil_P', color='#DF0071')
971.        #    current_ax.plot(BøTO[1][0:],  label = 'Oil_T', color='#00A030')
972.        Cal2 = Cals[0]
973.        Cal1 = Cals[1]
974.
975.        plt.plot(Cal1.GetKfactors(StreamName,'Oil')[50:], label ='Oil_Trad', color=Colors[
    0])
976.        plt.plot(Cal2.GetKfactors(StreamName,'Oil')[50:], label ='Oil_Para', color=Colors[
    1])
977.        plt.plot(Cal1.GetKfactors(StreamName,'Gas')[50:], label ='Gas_Trad', color=Colors[
    2])
978.        plt.plot(Cal2.GetKfactors(StreamName,'Gas')[50:], label ='Gas_Para', color=Colors[
    3])
979.        plt.plot(Cal1.GetKfactors(StreamName,'Water')[50:], label ='Water_Trad', color=Col
    ors[4])
980.        plt.plot(Cal2.GetKfactors(StreamName,'Water')[50:], label ='Water_Para', color=Col
    ors[6])
981.        #axes[0].plot(TBø[0][1][0:],  label = 'Oil_T', color='#00A030')
982.        plt.ylabel('K-factor',fontsize = 'x-large' )
983.        plt.xlabel('Syncronice runs across trials' ,fontsize = 'x-large')
984.        plt.legend(fontsize = 'x-large')
985.        plt.title(StreamName+' Comparison Calibration',fontsize = 'x-large')
986.
987.        plt.show()
988.
989.
990.    #Box plot comparrison eller violinplot noge sånt :)
991.    #def ViolinPlot(Calibrations,FigureSize = (10,10)):
992.    #    first = Calibrations[0]
993.    #    sec = Calibrations[1]
994.    #
995.    #
996.    #    plt.figure(figsize=FigureSize)
997.    #    ax = sns.violinplot(x='stream', y='k-factor', hue ='phase', data=self.BasisDF)
998.    #
999.    #    plt.show()
1000.
1001.
1002.   #unit testing of important functions
1003.   if (__name__ == '__main__'):
1004.
1005.       Frame = np.array([[1,1,0],[0,1,1],[1,0,1]])
```

```
1006.
1007.       baseT1 = np.array([1,1,2])
1008.       baseT2 = np.array([0.5,1,1.5])
1009.       baseT3 = np.array([0.25,0.5,0.75])
1010.       T1 = baseT1
1011.       T2 = baseT2
1012.       T3 = baseT3
1013.
1014.       x = np.linspace(1,1000,100)
1015.       for i in range(len(x)):
1016.          T1 = np.vstack((T1,baseT1*x[i]))
1017.          T2 = np.vstack((T2,baseT2*x[i]))
1018.          T3 = np.vstack((T3,baseT3*x[i]))
1019.
1020.       Trials = [[T1,T1,T1],[T2,T2,T2],[T3,T3,T3]]
1021.
1022.       Mm = GetM_m(Frame,Trials,0)
1023.
1024.       for i in range(len(Mm)):
1025.           M = Mm[i][0]
1026.           m = Mm[i][1]
1027.           print(str(M))
1028.           print(" = ")
1029.           print(str(m))
1030.
1031.
1032.       #Copy this from where it is used in the code. or just implement it as a function :
       )
1033.       Kfactors = []
1034.       for phase in range(3): # 3 phases Oil, Gas and Water in that order.
1035.           Mm = GetM_m(Frame,Trials,phase) #Gets a touple of the M matrix and m phase
1036.           first = True
1037.           for i in range(1,len(Mm)):
1038.               try:
1039.                   K_i = Solver(Mm[i])
1040.               except:
1041.                   print("Singular matrix at "+str(i)+"iteration")
1042.                   continue
1043.
1044.               if(first):
1045.                   K = K_i
1046.               else:
1047.                   K = np.vstack((K,K_i))
1048.               first = False
1049.
1050.           Kfactors.append(K)
1051.
1052.       for i in range(len(Kfactors[0][:,0])):
1053.           print(str(Kfactors[0][i][0])+" "+str(Kfactors[0][i][1])+" "+str(Kfactors[0][i]
       [2]))
1054.
1055.       print("""\n \n \n \n \n \n \n /n /n /n /n /n /n /n """)
1056.
1057.
1058.       for i in range(len(Kfactors[1][:,0])):
1059.           print(str(Kfactors[1][i][0])+" "+str(Kfactors[1][i][1])+" "+str(Kfactors[1][i]
       [2]))
1060.
1061.       print("""\n \n \n \n \n \n \n \n /n /n /n /n /n /n /n """)
1062.
1063.       for i in range(len(Kfactors[2][:,0])):
1064.           print(str(Kfactors[2][i][0])+" "+str(Kfactors[2][i][1])+" "+str(Kfactors[2][i]
       [2]))
1065.
```

6 Source Code - Calibration Statistics

Printed 12/05-2019

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  Created on Sat Apr 13 23:26:19 2019
4.
5.  @author: stig
6.  """
7.  import numpy as np
8.  import matplotlib.pyplot as plt
9.  from scipy.signal import find_peaks
10.
11.
12.  def SSE(Kfaktorer):
13.      """Finds the sum squared error within the dataset"""
14.
15.      totnum = len(Kfaktorer)
16.      Startnum = int(0.05*totnum)
17.      MaxUse = int(0.9*totnum) # Max use 90% of the values
18.      SSEs = []
19.      for i in range(MaxUse):
20.          ItersFactors = Kfaktorer[len(Kfaktorer)-Startnum-i:]
21.          mu = np.mean(ItersFactors) #The mean
22.          SSE = 0
23.          for j in range(len(ItersFactors)):
24.              SSE = SSE + (ItersFactors[j]-mu)**2
25.          SSEs.append(SSE)
26.      return SSEs
27.
28.  def DSSE(SSEs):
29.      """  Finds the differances of the sum squared error list  """
30.      DSSEs = []
31.      for i in range(1,len(SSEs)):
32.          DSSEs.append((SSEs[i-1]-SSEs[i])**2)
33.      return DSSEs
34.
35.
36.  def findStableValues(Kfaktorer,height=1.0e-06):
37.      """
38.      Finds the stable values from the end to a change, by finding slopes by differentiati
         on a Sum of squared errors, and use peak detection to detect
39.
40.      """
41.      SSEs = SSE(Kfaktorer)
42.      DSSEs = DSSE(SSEs)
43.
44.      peak, __ = find_peaks(np.asarray(DSSEs), height, distance=2.1)
45.
46.      if(len(peak)==0):
47.          peak, __ = find_peaks(np.asarray(DSSEs), 9.199606608655428e-07, distance=2.1)
48.          if(len(peak)==0):
49.              lim = int(0.8*len(Kfaktorer))
50.          else:
51.              lim = peak[0]
52.      else:
53.          lim = peak[0] #taking the first peak as the stopping point.
54.
55.
56.      Basis = Kfaktorer[len(Kfaktorer)-lim:]
57.
58.      return (Basis,lim)
59.
```

```
60.
61.
62.   def CalcRandUncert(Basis,rep=False):
63.       """
64.       This method numerically integrates a normal distrobution to probe the uncertanity of
           the Basis data.
65.
66.       """
67.       mean = np.mean(Basis)
68.       sigma = np.std(Basis)
69.       stdev = sigma #if you see this don't tell anyone.
70.       mu = mean
71.
72.       Coverage = 2 #Definint the coverage factor for integration limits
73.
74.       lower = mean-(Coverage*stdev)
75.       upper = mean+(Coverage*stdev)
76.
77.       steps = 10000 #numb of intervall segments for numerical integration of normal probab
          ility density distibution.
78.
79.       """      The Actual calculation and integration of the normal distrobution   """
80.       t = np.linspace(lower,upper,10000)
81.       dt = (t[1]-t[0])/steps
82.       f = 1/(np.sqrt(2*np.pi*(sigma**2)))*np.exp(-(t-
          mu)**2/(2*sigma**2)) #Created a normal distibution vector between +-
          2 std. deviations
83.
84.       X = f[0]*dt #inital datapoint with or without? i cannot decide, it is also a part of
           the first iteration in the for loop
85.       for i in range(1,len(t)):
86.           X = X + ((f[i-1]*f[i])/2)*dt #Numerically integrates a normalditrobution
87.
88.       if(rep):
89.           print("Mean="+str(mean)+"+-"+str(X*100)+"%  Sigma="+str(stdev))
90.
91.       return (mean,X,stdev)
92.
93.   def PlotStats(Kfaktorer,lim,Basis,title='',save=False):
94.
95.       Colors = ['#000000','#DF0071','#00A030','#EE7900','#777777',' #000028','#B9B9B9','#E
          6E6E6']
96.
97.       mu = np.mean(Basis)
98.       sigma = np.std(Basis)
99.       lower = mu-(4*sigma)
100.      upper = mu+(4*sigma)
101.
102.      steps = 1000
103.      t = np.linspace(lower,upper,steps)
104.      f = 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(t-mu)**2/(2*sigma**2))
105.
106.      rempart = 25
107.      ymin = np.min(Kfaktorer[rempart:])
108.      ymax = np.max(Kfaktorer[rempart:])
109.      fig = plt.figure()
110.      fig.suptitle(title)
111.      Trend = fig.add_axes([0.1,0.1,0.8,0.8])
112.      Prob = fig.add_axes([0.9,0.1,0.3,0.8])
113.      Trend.plot(Kfaktorer[rempart:],color=Colors[6])
114.      Trend.plot(np.arange(len(Kfaktorer)-lim-rempart,len(Kfaktorer)-
          rempart),Basis,color=Colors[1])
115.
116.      Prob.hist(Basis, bins=30, orientation = 'horizontal',color=Colors[2])
117.      Prob.plot(f,t, color=Colors[3])
118.      Prob.set_ylim(ymin,ymax)
```

```
119.      Trend.set_ylim(ymin,ymax)
120.      if(save):
121.          fig.savefig(title+".png")
122.      else:
123.          fig.show()
124.
125.
126.
127.
128.
129.  class stats():
130.      def __init__(self,Kfaktorer,name='',phase = '',PLen=-1):
131.
132.          self.name = name
133.          self.phase = phase
134.          self.Kfaktorer = Kfaktorer
135.          self.totStdev = np.std(Kfaktorer)
136.          self.totmean = np.mean(Kfaktorer)
137.          self.PeakDetLim = (self.totStdev**7)*100
138.          if (PLen == -1):
139.              self.Basis, self.lim = findStableValues(Kfaktorer,self.PeakDetLim)
140.          else:
141.              self.lim = PLen
142.              self.Basis = Kfaktorer[len(Kfaktorer)-lim:]
143.          self.mean , self.X ,self.stdev = CalcRandUncert(self.Basis)
144.          self.N = len(self.Basis)
145.
146.      def ReCalc(self,newlim):
147.          self.lim = newlim
148.          self.Basis = self.Kfaktorer[len(self.Kfaktorer)-self.lim:]
149.          self.mean , self.X ,self.stdev = CalcRandUncert(self.Basis)
150.          self.N = len(self.Basis)
151.
152.          self.plot()
153.          print(self.to_string())
154.
155.
156.      def plot(self):
157.          PlotStats(self.Kfaktorer,self.lim,self.Basis,self.name+" "+self.phase)
158.
159.      def plotDiag(self):
160.          """
161.          Diagnostics plot
162.          """
163.
164.          SSEs = SSE(self.Kfaktorer)
165.          DSSEs = DSSE(SSEs)
166.
167.          fig = plt.figure()
168.          axSSE = fig.add_axes()
169.          axDSSE = fig.add_axes()
170.          axSSE.plot(SSEs)
171.          axDSSE.plot(DSSEs)
172.
173.      def ResultVector(self):
174.          return np.array([self.mean,self.X*100,self.stdev,len(self.Basis)])
175.
176.      def to_string(self):
177.          return self.name+" "+self.phase+" mean = "+str(self.mean)+" ± "+str(self.X*100)+
          " % - St.dev ="+str(self.stdev)+" #Samples ="+str(len(self.Basis))
178.
179.
180.  if __name__ == "__main__":
181.      """ Unit testing of find and do statistics"""
182.      N_num = 1000 #må være partall
183.      x = np.linspace(0.5,-20,N_num)
```

```
184.      noise = (np.random.rand(N_num)-0.5)*0.03
185.      noisreduction = np.hstack((np.linspace(1.6,0.1,int(N_num/2)),0.1*np.ones(int(N_num/2
          ))))
186.      noise = np.multiply(noise,noisreduction)
187.      cumunoise = np.zeros(len(noise))
188.      for i in range(1,len(noise)):
189.          cumunoise[i] =cumunoise[i-1]+noise[i]
190.      y = 1+0.5*np.exp(10*x)/100+cumunoise#+(0.01*np.sin(x))
191.      #y[int(0.3*1000)] = y[int(0.7*1000)]*1.05
192.
193.      plt.plot(y)
194.      plt.title('Randomly generated k-factor development')
195.      plt.show()
196.      SSEs = SSE(y)
197.      plt.plot(SSEs)
198.      plt.title('SSE')
199.      plt.show()
200.      DSSEs = DSSE(SSEs)
201.      plt.plot(DSSEs)
202.      plt.title('DSSE')
203.      plt.show()
204.      plt.semilogy(DSSEs)
205.      plt.title('DSSE - logarithmic')
206.      plt.show()
207.      std = np.std(y)
208.      (Basis,lim) = findStableValues(y,std**7*100)
209.      print("lim = "+str(lim))
210.      plt.plot(Basis)
211.      plt.show()
212.      PlotStats(y,lim,Basis)
213.      CalcRandUncert(Basis,rep=True)
214.      test = stats(y,'unit testing')
215.      print(test.to_string())
216.      test.plot()
217.      test.ReCalc(259)
218.
```

1 Appendix E – Calibration Execution and Results January

Contents

This appendix will have the results of the performed calibrations both Traditional, synthetic and perform a comparison. There are two data sets available for this comparison one from January 2019 and one from late march the same year. This Appendix will cover the execution of these test and give the data and result of the traditional and synthetic calibration. And gives insight to the accuracy of the parallel algorithm compared to a traditional method.

1 Appendix E – Calibration Execution and Results January 1

Contents 2

2 January – Traditional vs Synthetic 3

2.1 Synthetic Parallel calibration 4

2.1.1 Trial Plots 5

2.1.2 Statistics 6

2.2 Traditional calibration 8

2.2.1 Trial Plots 8

2.2.2 Statistics 10

2.3 Comparison 13

3 Compare result to real calibration 18

2 January – Traditional vs Synthetic

Traditional calibration was carried out in January 2019, this chapter will perform a traditional and synthetic parallel calibration. Also, worth to note that the Multi-phase meter on Bøyla was changed to another model, and in the march test later in this appendix a new installed multi-phase meter will be in operations.

Figure 1 shows the first lines of code, considering the implementation of standard methods, and formatting the output as well as getting the parallel calibration libraries an a Cognite client and getting the time windows when the traditional calibration was carried out.

In [35]:
```
%%javascript
IPython.OutputArea.auto_scroll_threshold = 9999;
```

In [1]:
```
import numpy as np
import pandas as pd
from datetime import datetime
from datetime import timedelta
from matplotlib import pyplot as plt
%matplotlib inline
```

In [2]:
```
from ParralellCalibration import *
```

In [4]:
```
from cognite import CogniteClient
cdp = CogniteClient()
```

In [5]:
```
from JanuarTimes import *
```

```
(datetime.datetime(2019, 1, 27, 5, 18, 30), datetime.datetime(2019, 1, 27, 13,
8, 30))
(datetime.datetime(2019, 1, 26, 5, 22, 30), datetime.datetime(2019, 1, 26, 13,
35))
(datetime.datetime(2019, 1, 25, 5, 38), datetime.datetime(2019, 1, 25, 13, 2))
```

In [6]:
```
Duration = timedelta(minutes=400)
```

Figure 1 - Initial libraries and locating time windows for traditional calibration

2.1 Synthetic Parallel calibration

In [7]:
```
SynteticT1 = []
SynteticT1.append(MulitPhaseStream("Bøyla",Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp))
SynteticT1.append(MulitPhaseStream("Vilje",Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R1Bøyla =SeparatorStreams(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp)
R1Vilje =SeparatorStreams(Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp)
Ref1 = R1Vilje
Ref1.OilMass.data['cumulative'] = AddStreams(R1Vilje.OilMass.data,R1Bøyla.OilMass.data)
Ref1.GasMass.data['cumulative'] = AddStreams(R1Vilje.GasMass.data,R1Bøyla.GasMass.data)
Ref1.WtrMass.data['cumulative'] = AddStreams(R1Vilje.WtrMass.data,R1Bøyla.WtrMass.data)

SynteticT1.append(Ref1)
```

In [8]:
```
SynteticT2 = []

SynteticT2.append(MulitPhaseStream("Vilje",Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp))
SynteticT2.append(MulitPhaseStream("Volund",Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R2Vilje =SeparatorStreams(Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp)
R2Volund =SeparatorStreams(Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp)
Ref2 = R2Volund
Ref2.OilMass.data['cumulative'] = AddStreams(R2Volund.OilMass.data,R2Vilje.OilMass.data)
Ref2.GasMass.data['cumulative'] = AddStreams(R2Volund.GasMass.data,R2Vilje.GasMass.data)
Ref2.WtrMass.data['cumulative'] = AddStreams(R2Volund.WtrMass.data,R2Vilje.WtrMass.data)
SynteticT2.append(Ref2)
```

In [9]:
```
SynteticT3 = []

SynteticT3.append(MulitPhaseStream("Bøyla",Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp))
SynteticT3.append(MulitPhaseStream("Volund",Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R3Bøyla =SeparatorStreams(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp)
R3Volund =SeparatorStreams(Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp)
Ref3 = R3Bøyla
Ref3.OilMass.data['cumulative'] = AddStreams(R3Bøyla.OilMass.data,R3Volund.OilMass.data)
Ref3.GasMass.data['cumulative'] = AddStreams(R3Bøyla.GasMass.data,R3Volund.GasMass.data)
Ref3.WtrMass.data['cumulative'] = AddStreams(R3Bøyla.WtrMass.data,R3Volund.WtrMass.data)

SynteticT3.append(Ref3)
```

In [13]:
```
Syntetic = ParralellCalibration([SynteticT1,SynteticT2,SynteticT3])
```

Figure 2 - Initial code and Synthetic parallel calibration

Figure 2 shows the initial code, then processed to get the time windows for traditional calibration. Then it creates a synthetic trial dataset based on the streams in question, for each trial the streams remain the same, but the reference streams (separators streams) are added together.

Figure 3 shows the development of k-factors over the elapsed time of the trials.

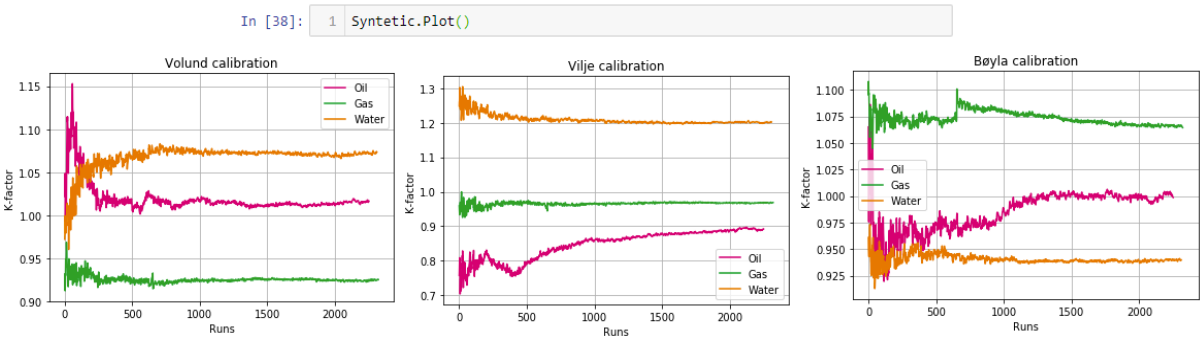


Figure 3 - Synthetic K-factor development

Since this is a synthetic dataset, when gauging the process stability during these trials, this can be seen in Figure 9 in the traditional method, since the synthetic data does nothing with the intensive variables in the system, this won't be representable of the real states, due to the higher the flow is through the separator, both the temperature and pressure should be higher.

2.1.1 Trial Plots

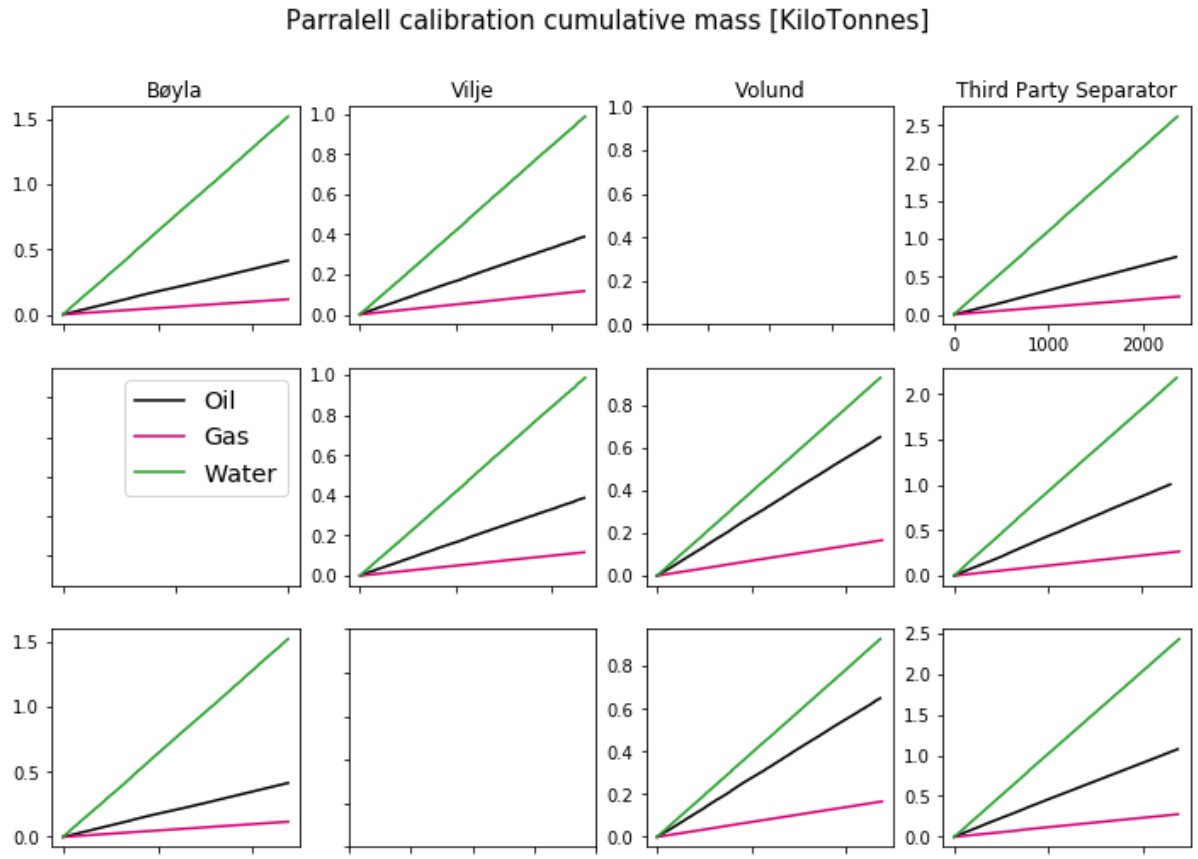


Figure 4 – synthetic cumulative mass matrix plot

Figure 4 shows the data going into the parallel calibration solver, where the synthetic here is the Third party separator as explained in the code in Figure 2.

2.1.2 Statistics

The result of the calibration is shown in Figure 5, which is based on the statistical basis shown in Figure 6.

```
In [39]: Syntetic.StatsDF
```

Out[39]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Bøyla	Oil	0.999605	0.541444	0.002593	1060.0
	Gas	1.066666	1.335782	0.001051	500.0
	Water	0.939100	1.152746	0.001218	1500.0
Vilje	Oil	0.882737	0.199537	0.007036	935.0
	Gas	0.966499	0.490514	0.002862	1897.0
	Water	1.200562	0.784741	0.001789	1124.0
Volund	Oil	1.014309	0.412972	0.003399	1947.0
	Gas	0.925355	1.415619	0.000992	1000.0
	Water	1.071862	0.825541	0.001701	1327.0

Figure 5- Resulting data frame of synthetic calibration

2 January – Traditional vs Synthetic

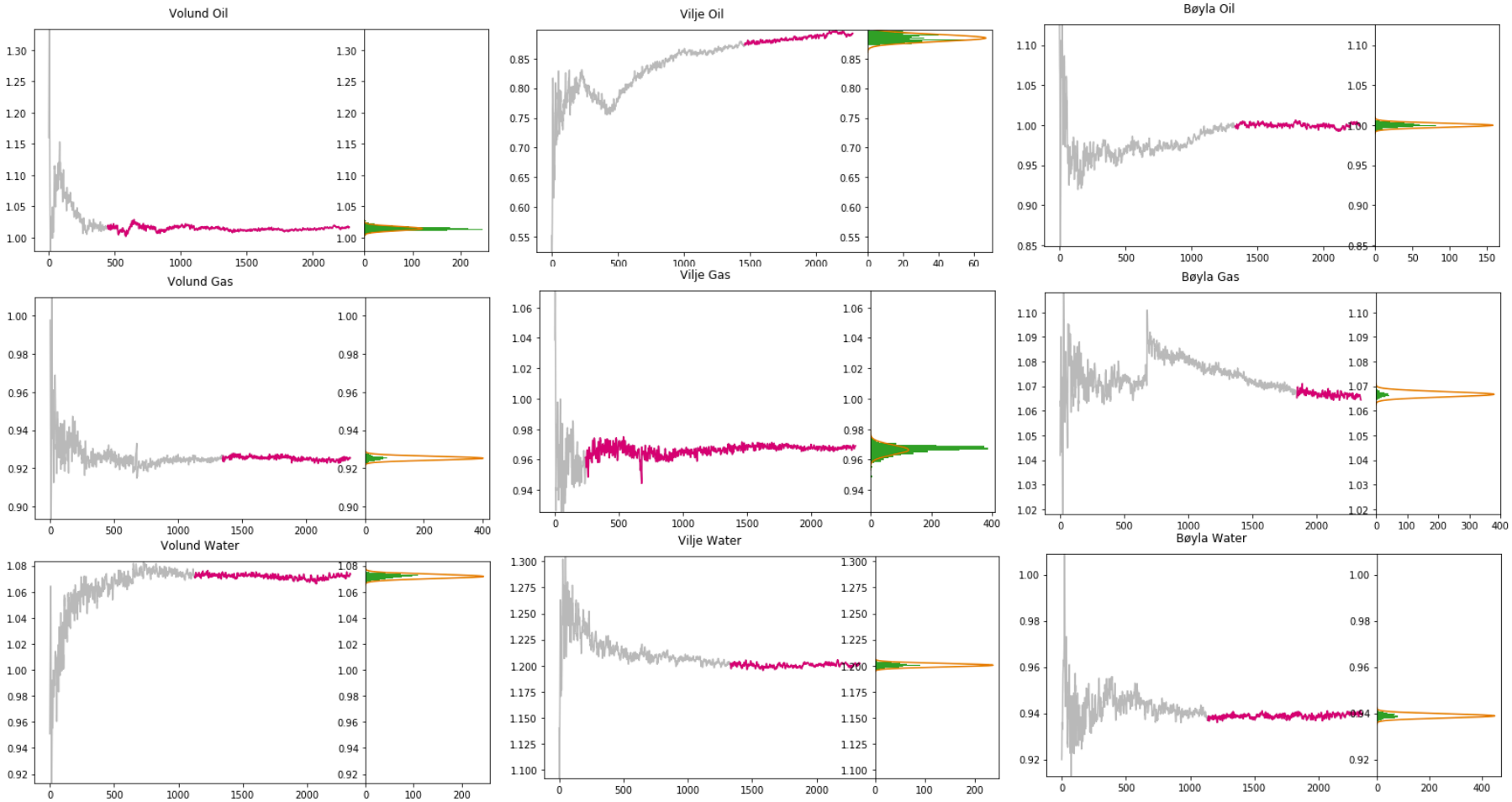


Figure 6 - Statistical basis form k-factor development synthetic calibration, with vertical histogram with number of samples and probability density

2.2 Traditional calibration

```
In [10]:
1 Trial1 = CreateTrial(["Volund"],(Volund_timewindow[0],Volund_timewindow[0]+Duration),cdp)
2 Trial2 = CreateTrial(["Vilje"],(Vilje_timewindow[0],Vilje_timewindow[0]+Duration),cdp)
3 Trial3 = CreateTrial(["Bøyla"],(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration),cdp)

Creating Volund Stream
Creating Separator Streams
Trial Finnished
Creating Vilje Stream
Creating Separator Streams
Trial Finnished
Creating Bøyla Stream
Creating Separator Streams
Trial Finnished

In [11]:
1 TradCal = ParralellCalibration([Trial1,Trial2,Trial3])
```

Figure 7 - Traditional calibration performed

Figure 7 Shows the execution of a traditional calibration but uses the parallel calibration algorithm, but the results are the same, where the trial matrix is a diagonal matrix with only non-zero values on the diagonal, the inverse of the diagonal matrix then becomes the reciprocals of the values in each of the diagonal line. All in all, the use of the parallel calibration class to perform a traditional calibration can be and is done.

Figure 8 Shows the resulting k-factor development through the trials.

```
In [40]:
1 TradCal.Plot()
```

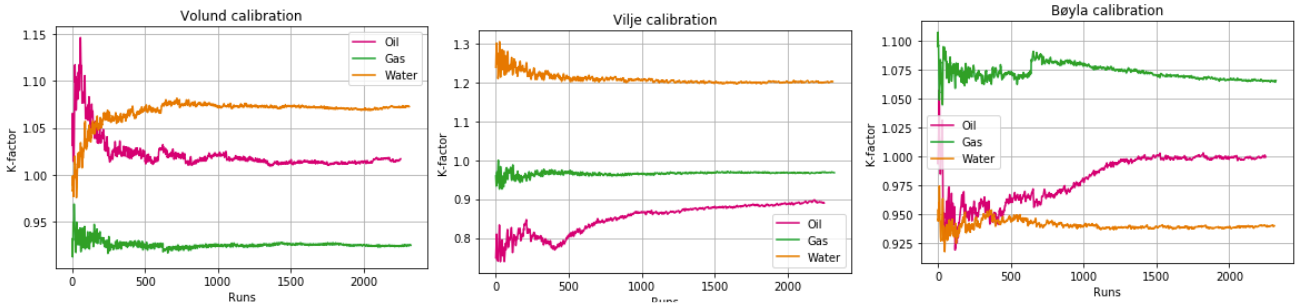


Figure 8 – Resulting k-factors over the Traditional calibration

2.2.1 Trial Plots

Figure 9 shows the intensive variables of the trials. Figure 10 shows the cumulative matrix plot of the accumulated mass, this plot is the values over the different runs going into the solver

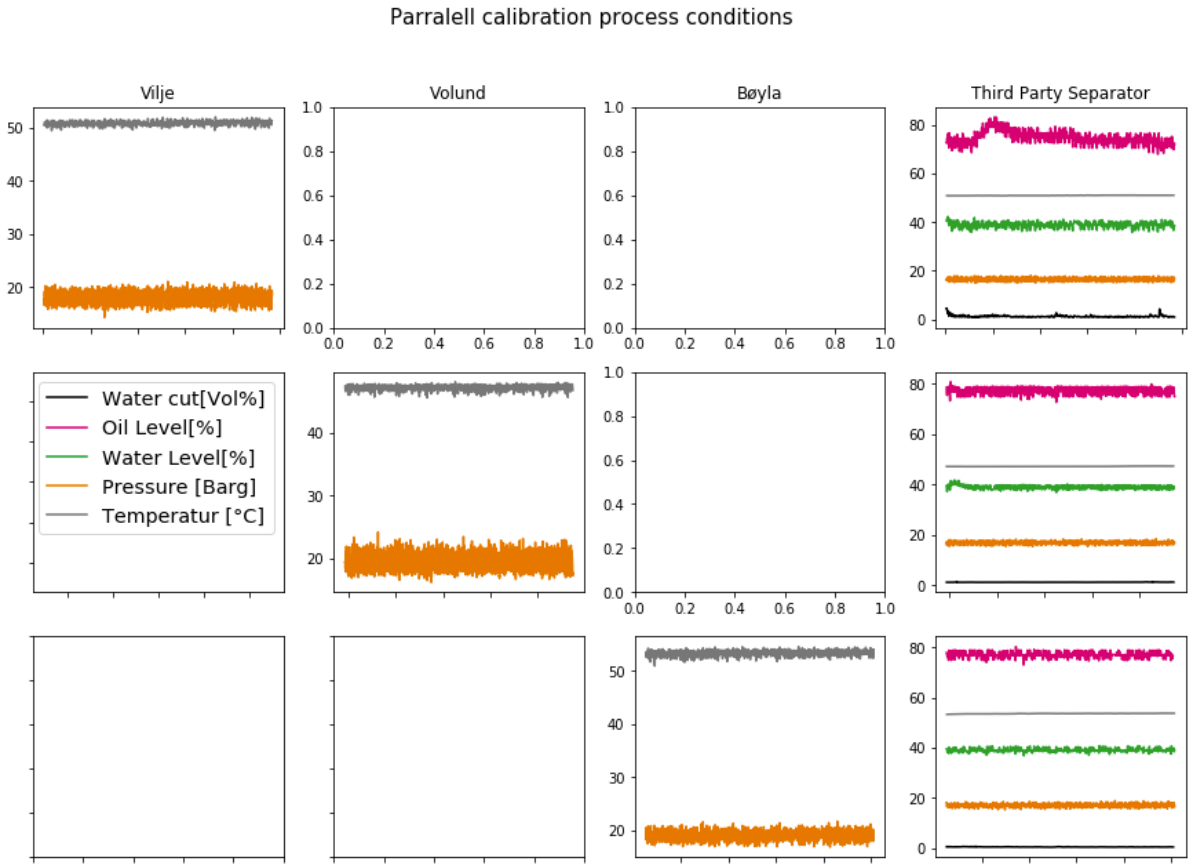


Figure 9 - Intensive variables of data set

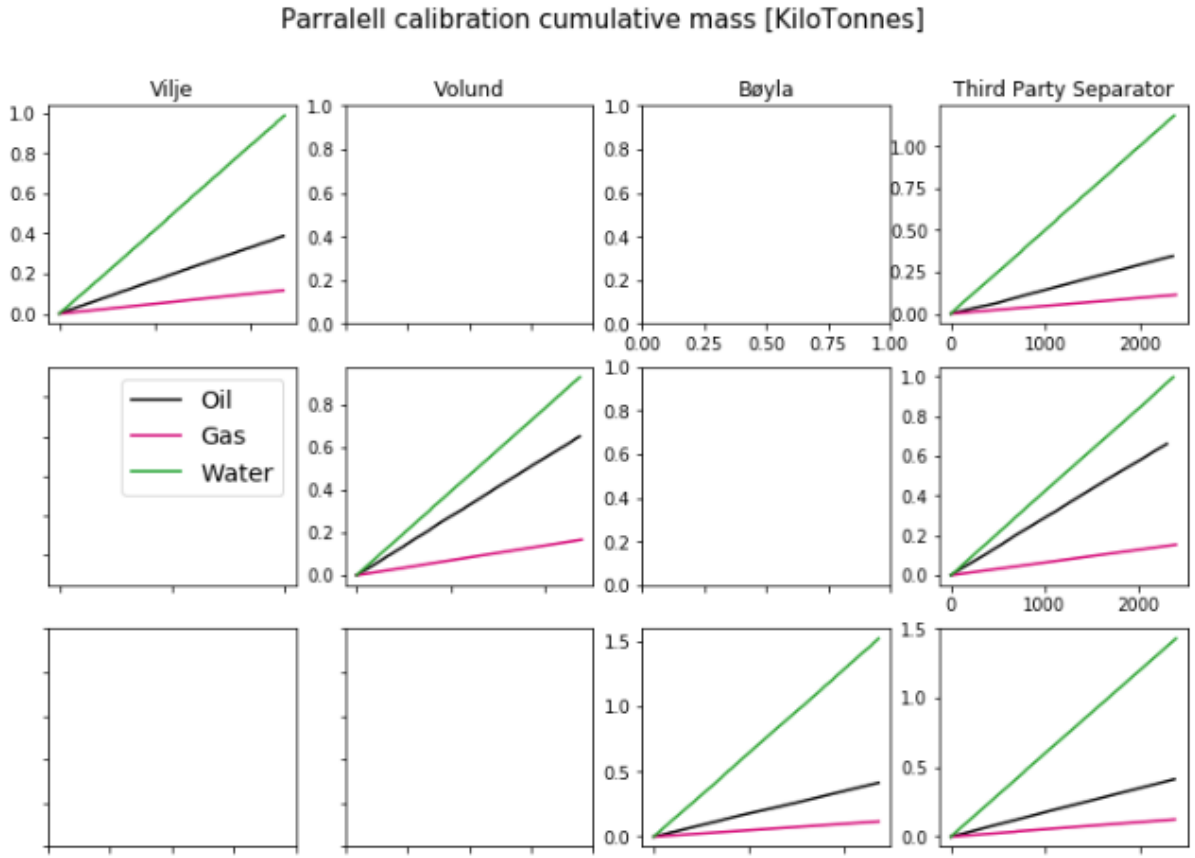


Figure 10 - Cumulative matrix plot

2.2.2 Statistics

Figure 11 shows the resulting values for each stream and each phase based on the data basis shown in Figure 12.

```
In [38]: TradCal.StatsDF
```

Out[38]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Vilje	Oil	0.883457	0.232006	0.006051	958.0
	Gas	0.966498	0.570060	0.002463	1897.0
	Water	1.200458	0.784854	0.001789	1122.0
Volund	Oil	1.016419	0.325407	0.004314	1923.0
	Gas	0.925481	1.414976	0.000992	1000.0
	Water	1.072090	0.942964	0.001489	1394.0
Bøyla	Oil	0.998741	0.990563	0.001417	996.0
	Gas	1.066596	1.415145	0.000992	500.0
	Water	0.938869	1.825493	0.000769	1000.0

Figure 11 - Resulting data frame of calibration

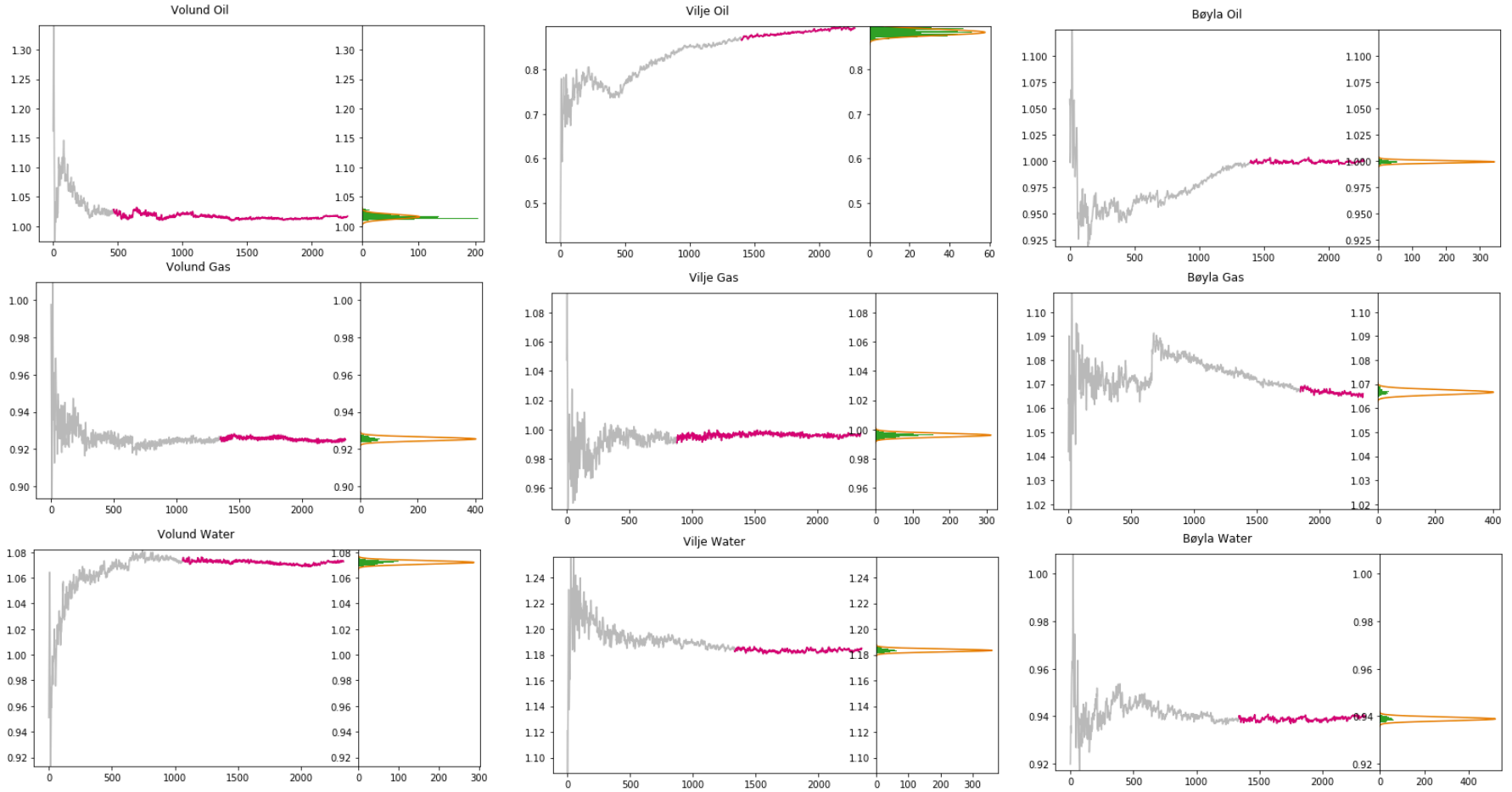


Figure 12 - Statistical basis form k-factor development traditional calibration, with vertical histogram with number of samples and probability density

2.3 Comparison

The result of both calibration compared towards each other for each stream is shown in Figure 13 for Volund, Figure 14 for Vilje and Figure 15 for Bøyla multi-phase flow meters. Just form these figures the result of both methods seam to replicate each other very well and that the parallel calibration method based on synthetic data works very well. A print of the data frames of the differences is also shown in Figure 16. And is collected in Table 1.

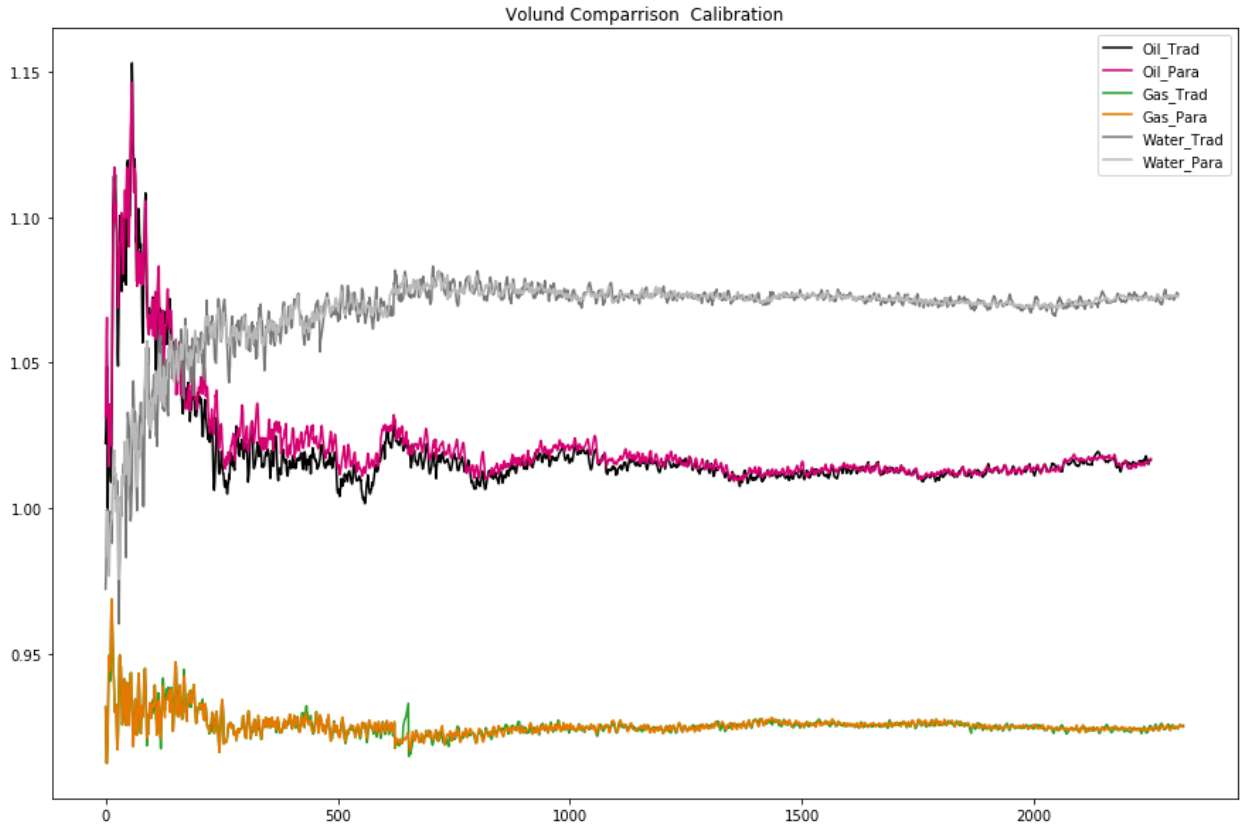


Figure 13 - Volund calibration comparrison

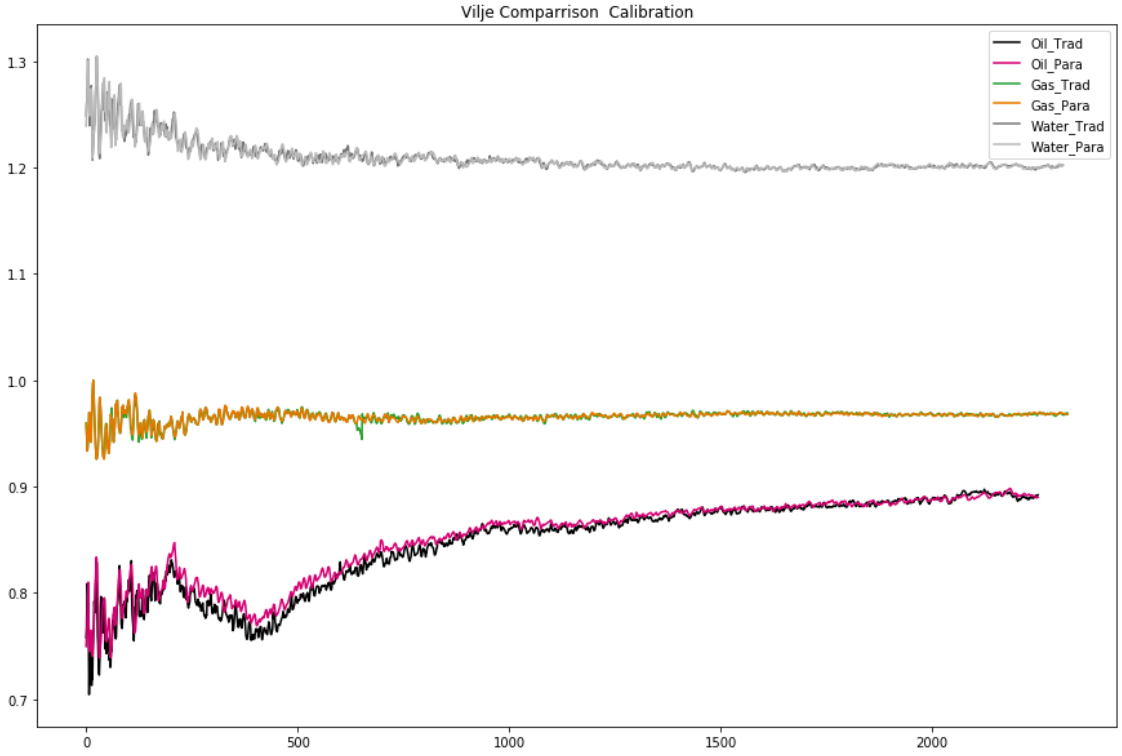


Figure 14- Vilje calibration comparrison

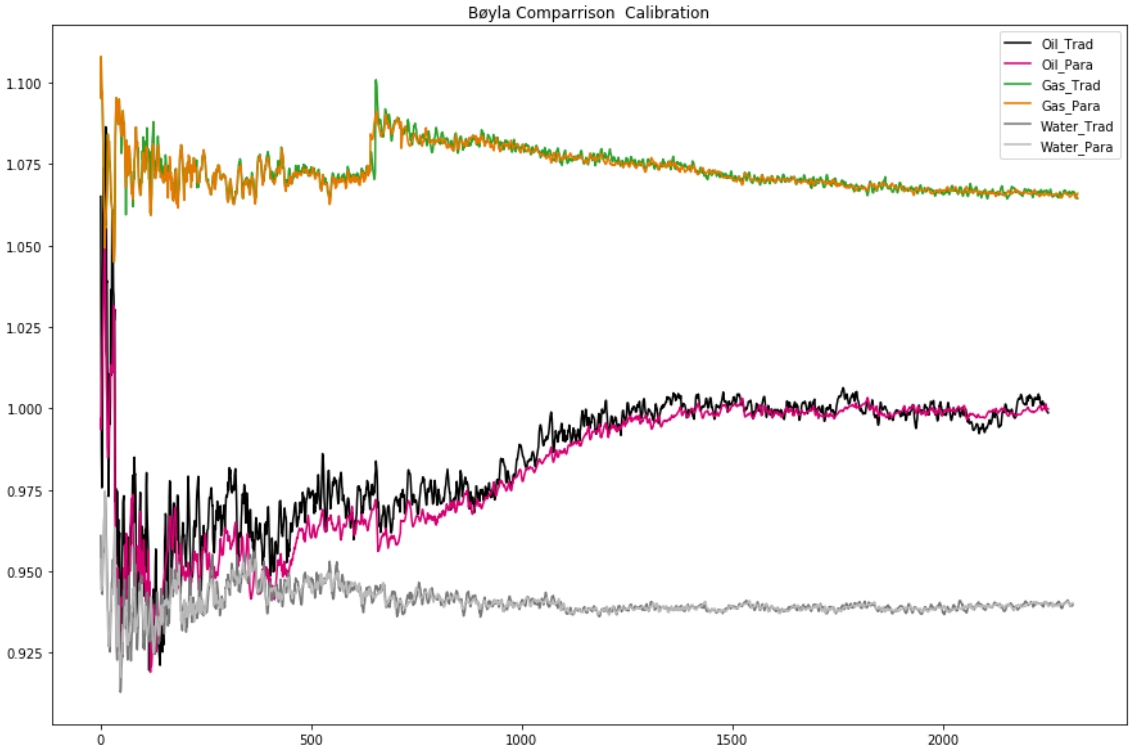


Figure 15 - Bøyla calibration comparrison

```
In [40]: (TradCal.StatsDF.loc['Bøyla']-Syntetic.StatsDF.loc['Bøyla'])
```

Out[40]:

Phase	mean	uncert	St.dev	SampleSize
Oil	-0.000864	0.449119	-0.001176	-64.0
Gas	-0.000069	0.079363	-0.000059	0.0
Water	-0.000230	0.672747	-0.000449	-500.0

```
In [41]: (TradCal.StatsDF.loc['Vilje']-Syntetic.StatsDF.loc['Vilje'])
```

Out[41]:

Phase	mean	uncert	St.dev	SampleSize
Oil	0.000720	0.032468	-9.846128e-04	23.0
Gas	-0.000001	0.079546	-3.993710e-04	0.0
Water	-0.000104	0.000113	-2.580672e-07	-2.0

```
In [42]: (TradCal.StatsDF.loc['Volund']-Syntetic.StatsDF.loc['Volund'])
```

Out[42]:

Phase	mean	uncert	St.dev	SampleSize
Oil	0.002110	-0.087565	9.147768e-04	-24.0
Gas	0.000126	-0.000643	4.508911e-07	0.0
Water	0.000228	0.117422	-2.117613e-04	67.0

Figure 16 - Print of differences between Traditional and synthetic calibration

2 January – Traditional vs Synthetic

Table 1 - Comparison of Traditional an Synthetic parallel

January Calibration		Traditional				Synthetic Parallel				Differences between Synthetic and Traditional (ref)			
Stream	Phase	mean	random uncertainty [%]	Standard deviation	sample size	mean	random uncertainty [%]	Standard deviation	sample size	deviation on k-factor	Deviation	diff rand. uncertainty	stability (St.dev-St.dev)
Bøyla	Oil	0.99874	0.99	1.42E-03	996	0.99961	0.54	2.59E-03	1060	8.64E-04	0.09%	-0.449	0.001
	Gas	1.06660	1.42	9.92E-04	500	1.06667	1.34	1.05E-03	500	7.00E-05	0.01%	-0.079	0.000
	Water	0.93887	1.83	7.69E-04	1000	0.93910	1.15	1.22E-03	1500	2.31E-04	0.02%	-0.673	0.000
Volund	Oil	1.01642	0.33	4.31E-03	1923	1.01431	0.41	3.40E-03	1947	-2.11E-03	-0.21%	0.088	-0.001
	Gas	0.92548	1.41	9.92E-04	1000	0.92536	1.42	9.92E-04	1000	-1.26E-04	-0.01%	0.001	0.000
	Water	1.07209	0.94	1.49E-03	1394	1.07186	0.83	1.70E-03	1327	-2.28E-04	-0.02%	-0.117	0.000
Vilje	Oil	0.88346	0.23	6.05E-03	958	0.88274	0.20	7.04E-03	935	-7.20E-04	-0.08%	-0.032	0.001
	Gas	0.96650	0.57	2.46E-03	1897	0.96650	0.49	2.86E-03	1897	1.00E-06	0.00%	-0.080	0.000
	Water	1.20046	0.78	1.79E-03	1122	1.20056	0.78	1.79E-03	1124	1.04E-04	0.01%	0.000	0.000

3 Compare result to real calibration

Table 2 below compare the different result to the actual accepted k-factors from the metering system on Alvheim, and this is subsequently plotted in Figure 17, Figure 18 and Figure 19 for each respective multi-phase meter streams.

Table 2 - Result compared to real result

January Calibration	Phase	Current metering system	Algorithm	
			Traditional	Synthetic parallel
Bøyla	Oil	0.97919	0.99874	0.99961
	Gas	1.01439	1.0666	1.06667
	Water	0.93353	0.93887	0.9391
Vilje	Oil	0.86164	0.88346	0.88274
	Gas	0.96043	0.9665	0.9665
	Water	1.15805	1.20046	1.20056
Volund	Oil	0.98797	1.01642	1.01431
	Gas	0.90674	0.92548	0.92536
	Water	1.04478	1.07209	1.07186

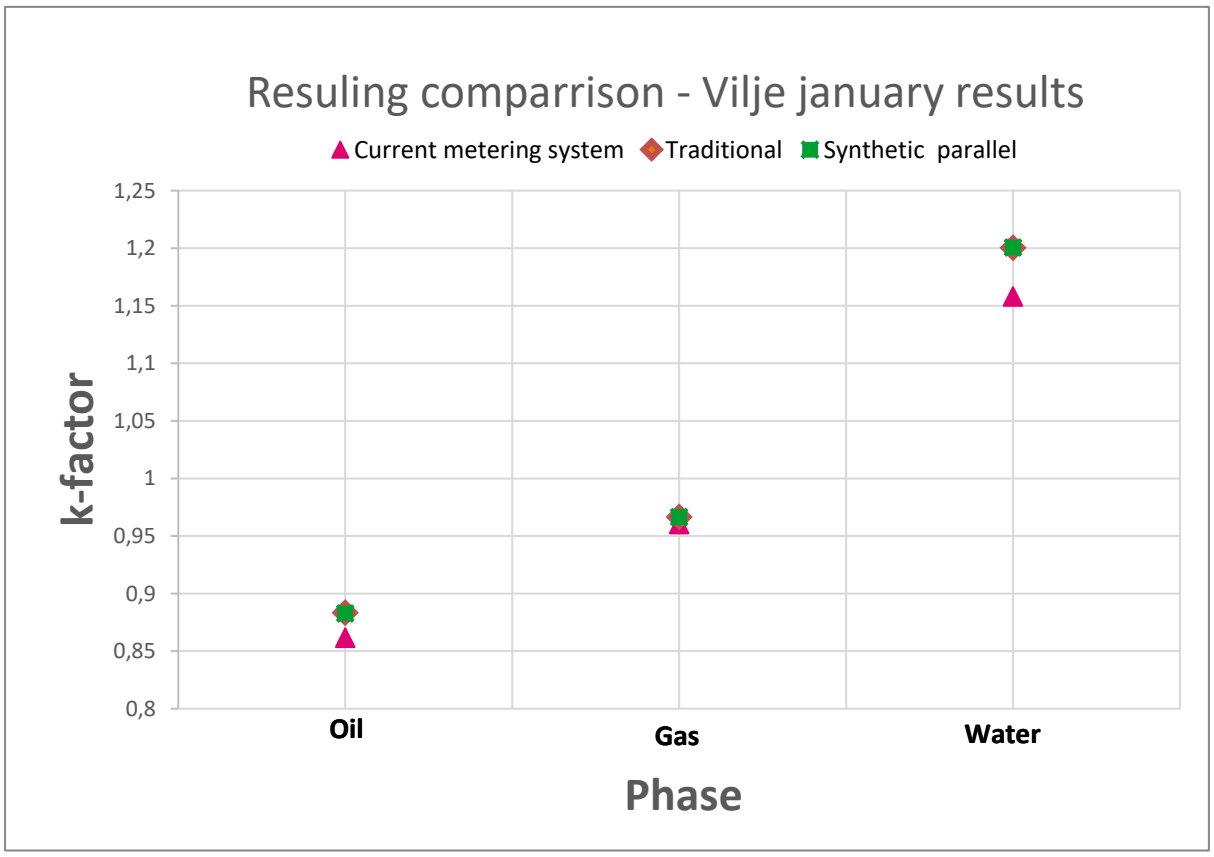


Figure 17 - calibration result comparison Vilje January

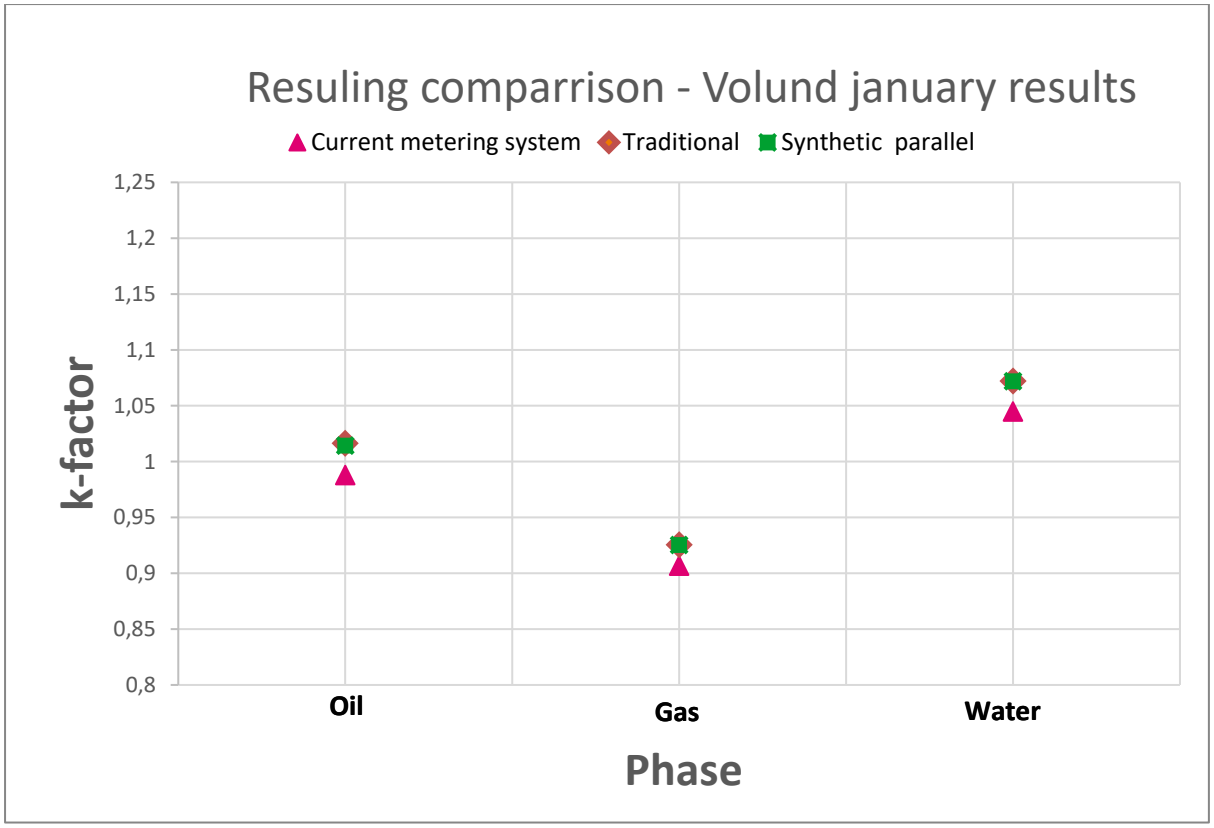


Figure 18 - calibration result comparison Volund January

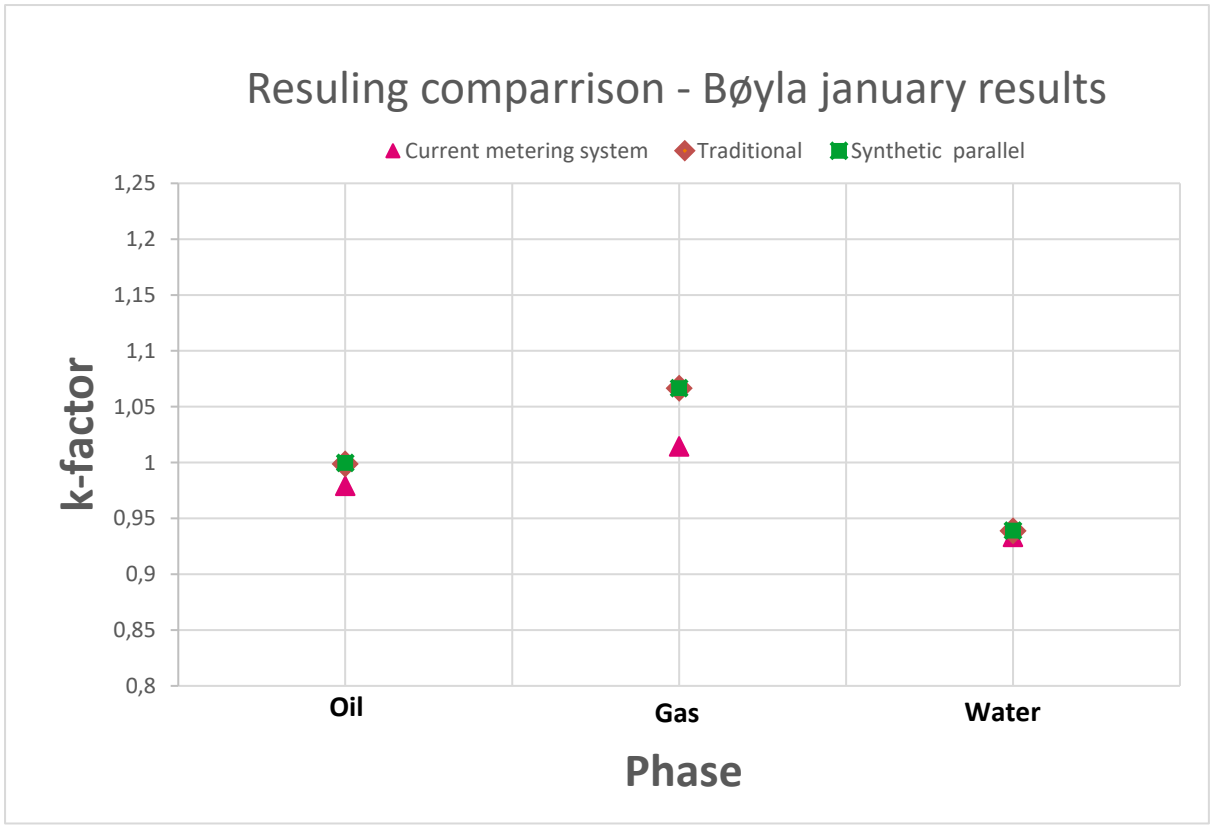


Figure 19 - calibration result comparison Bøyla January

1 Appendix F Calibration execution and result April

Contents

This appendix will have the results of the performed calibrations both Traditional, synthetic and perform a comparison. There are two data sets available for this comparison one from January 2019 and one from late march the same year. This Appendix will cover the execution and result of parallel calibrations in both the start and end of April and compare the results.

1 Appendix F Calibration execution and result April 1

2 Traditional vs Synthetic 3

2.1 Synthetic Parallel calibration 4

2.1.1 Trial Plots 5

2.1.2 Statistics 6

2.2 Traditional calibration 8

2.2.1 Trial Plots 8

2.2.2 Statistics 10

2.3 Comparison 13

3 2x2x2 parallel calibration 15

4 3x2x2 parallel calibration 21

5 One month later 3x2x2 28

6 Result 33

2 Traditional vs Synthetic

Traditional calibration was carried out in late March 2019, this chapter will perform a traditional and synthetic parallel calibration, this is also the initial calibration of a newly installed multiphase meter on Bøyla.

Figure 1 shows the first lines of code, considering the implementation of standard methods, and formatting the output as well as getting the parallel calibration libraries and a Cognite client and getting the time windows when the traditional calibration was carried out.

In [35]:
```
%%javascript
IPython.OutputArea.auto_scroll_threshold = 9999;
```

In [1]:
```
import numpy as np
import pandas as pd
from datetime import datetime
from datetime import timedelta
from matplotlib import pyplot as plt
%matplotlib inline
```

In [2]:
```
from ParralellCalibration import *
```

In [4]:
```
from cognite import CogniteClient
cdp = CogniteClient()
```

In [5]:
```
from JanuarTimes import *
```

```
(datetime.datetime(2019, 1, 27, 5, 18, 30), datetime.datetime(2019, 1, 27, 13,
8, 30))
(datetime.datetime(2019, 1, 26, 5, 22, 30), datetime.datetime(2019, 1, 26, 13,
35))
(datetime.datetime(2019, 1, 25, 5, 38), datetime.datetime(2019, 1, 25, 13, 2))
```

In [6]:
```
Duration = timedelta(minutes=400)
```

Figure 1 - Initial libraries and locating time windows for traditional calibration

2.1 Synthetic Parallel calibration

In [7]:
```
SynteticT1 = []

SynteticT1.append(MulitPhaseStream("Bøyla",Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp))
SynteticT1.append(MulitPhaseStream("Vilje",Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R1Bøyla =SeparatorStreams(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp)
R1Vilje =SeparatorStreams(Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp)
Ref1 = R1Vilje
Ref1.OilMass.data['cumulative'] = AddStreams(R1Vilje.OilMass.data,R1Bøyla.OilMass.data)
Ref1.GasMass.data['cumulative'] = AddStreams(R1Vilje.GasMass.data,R1Bøyla.GasMass.data)
Ref1.WtrMass.data['cumulative'] = AddStreams(R1Vilje.WtrMass.data,R1Bøyla.WtrMass.data)

SynteticT1.append(Ref1)
```

In [8]:
```
SynteticT2 = []

SynteticT2.append(MulitPhaseStream("Vilje",Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp))
SynteticT2.append(MulitPhaseStream("Volund",Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R2Vilje =SeparatorStreams(Vilje_timewindow[0],Vilje_timewindow[0]+Duration,cdp)
R2Volund =SeparatorStreams(Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp)
Ref2 = R2Volund
Ref2.OilMass.data['cumulative'] = AddStreams(R2Volund.OilMass.data,R2Vilje.OilMass.data)
Ref2.GasMass.data['cumulative'] = AddStreams(R2Volund.GasMass.data,R2Vilje.GasMass.data)
Ref2.WtrMass.data['cumulative'] = AddStreams(R2Volund.WtrMass.data,R2Vilje.WtrMass.data)
SynteticT2.append(Ref2)
```

In [9]:
```
SynteticT3 = []

SynteticT3.append(MulitPhaseStream("Bøyla",Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp))
SynteticT3.append(MulitPhaseStream("Volund",Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp))

#Syntetic Combination of separator streams
R3Bøyla =SeparatorStreams(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration,cdp)
R3Volund =SeparatorStreams(Volund_timewindow[0],Volund_timewindow[0]+Duration,cdp)
Ref3 = R3Bøyla
Ref3.OilMass.data['cumulative'] = AddStreams(R3Bøyla.OilMass.data,R3Volund.OilMass.data)
Ref3.GasMass.data['cumulative'] = AddStreams(R3Bøyla.GasMass.data,R3Volund.GasMass.data)
Ref3.WtrMass.data['cumulative'] = AddStreams(R3Bøyla.WtrMass.data,R3Volund.WtrMass.data)

SynteticT3.append(Ref3)
```

In [10]:
```
Syntetic = ParralellCalibration([SynteticT1,SynteticT2,SynteticT3])
```

Figure 2 - Initial code and Synthetic parallel calibration

Figure 2 shows the initial code, then processed to get the time windows for traditional calibration. Then it creates a synthetic trial dataset based on the streams in question, for each trial the streams remain the same, but the reference streams (separators streams) are added together.

Figure 3 shows the development of k-factors over the elapsed time of the synthetic trials.

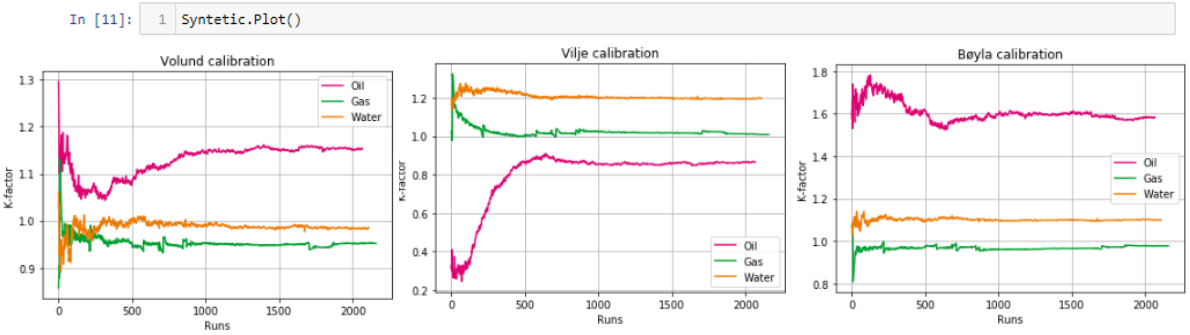


Figure 3 - Synthetic K-factor development

Since this is a synthetic dataset, when gauging the process stability during these trials, this can be seen in Figure 9 in the traditional method, since the synthetic data does nothing with the intensive variables in the system, this won't be representable of the real states, due to the higher the flow is through the separator, both the temperature and pressure should be higher.

2.1.1 Trial Plots

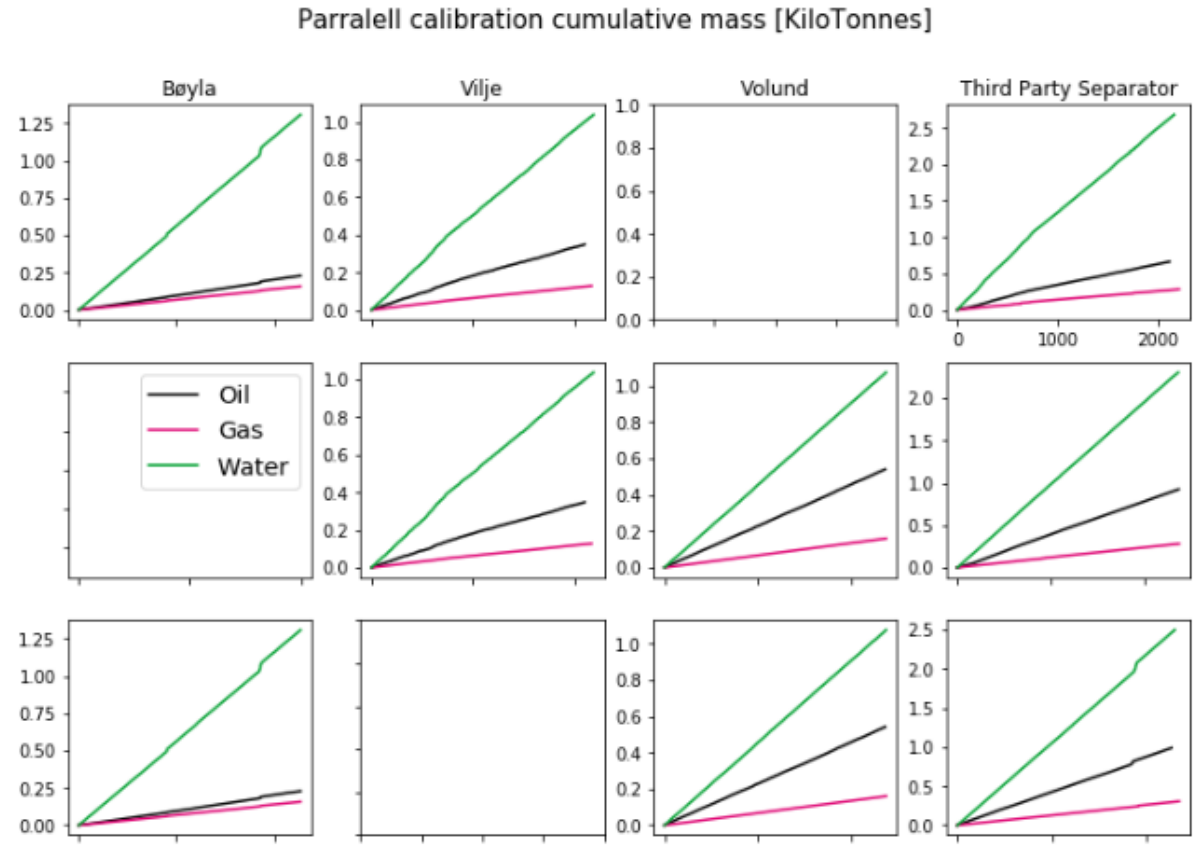


Figure 4 – synthetic cumulative mass matrix plot

Figure 4 shows the data going into the parallel calibration solver, where the synthetic here is the 3rd party separator as explained in the code in Figure 2.

2.1.2 Statistics

The result of the calibration is shown in Figure 5 which is based on the statistical basis shown Figure 6.

```
In [39]:  1  Syntetic.StatsDF
```

Out[39]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Bøyla	Oil	1.591172	0.128115	0.010958	1186.0
	Gas	0.977627	1.775918	0.000791	270.0
	Water	1.097893	0.710340	0.001976	1200.0
Vilje	Oil	0.863048	0.130138	0.010788	1584.0
	Gas	1.016802	0.314180	0.004468	1038.0
	Water	1.199346	0.374610	0.003748	1378.0
Volund	Oil	1.151642	0.342508	0.004099	1127.0
	Gas	0.951682	1.522929	0.000922	300.0
	Water	0.984497	1.176917	0.001193	400.0

Figure 5- Resulting data frame of synthetic calibration

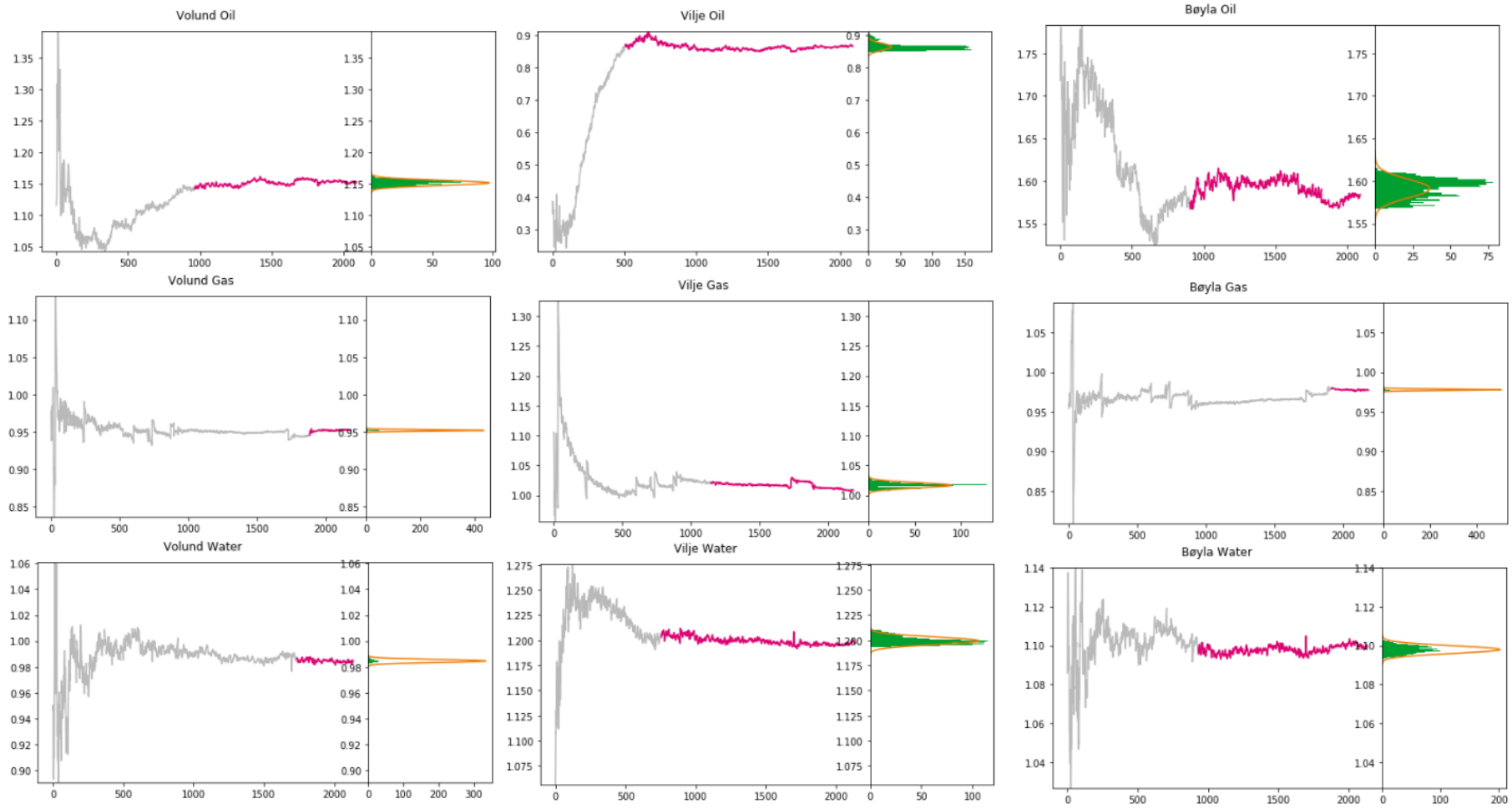


Figure 6 - Statistical basis of synthetic calibration

2.2 Traditional calibration

```
In [13]:
1 Trial1 = CreateTrial(["Volund"],(Volund_timewindow[0],Volund_timewindow[0]+Duration),cdp)
2 Trial2 = CreateTrial(["Vilje"],(Vilje_timewindow[0],Vilje_timewindow[0]+Duration),cdp)
3 Trial3 = CreateTrial(["Bøyla"],(Bøyla_timewindow[0],Bøyla_timewindow[0]+Duration),cdp)

Creating Volund Stream
Creating Separator Streams
Trial Finnished
Creating Vilje Stream
Creating Separator Streams
Trial Finnished
Creating Bøyla Stream
Creating Separator Streams
Trial Finnished

In [14]:
1 TradCal = ParralellCalibration([Trial2,Trial1,Trial3])
```

Figure 7 - Traditional calibration performed

Figure 7 Shows the execution of a traditional calibration but uses the parallel calibration algorithm, but the results are the same, where the trial matrix is a diagonal matrix with only non-zero values on the diagonal, the inverse of the diagonal matrix then becomes the reciprocals of the values in each of the diagonal line. All in all, the use of the parallel calibration class to perform a traditional calibration can be and is done.

Figure 8 Shows the resulting k-factor development through the trials.

```
In [15]:
1 TradCal.Plot()
```

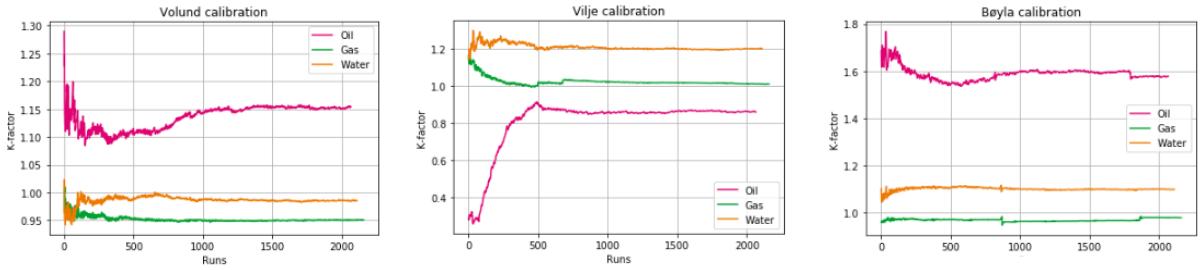


Figure 8 – Resulting k-factors over the Traditional calibration

2.2.1 Trial Plots

Figure 9 shows the intensive variables of the trials, with a blue square around the separator conditions on the Vilje trial; during the Vilje trial the separator conditions where not stable there where significant changes to the oil level and peaks of high water cut, which are not nessasery beneficial for a parallel calibration Figure 10 shows the cumulative matrix plot of the accumulated mass, this plot is the values over the different runs going into the solver

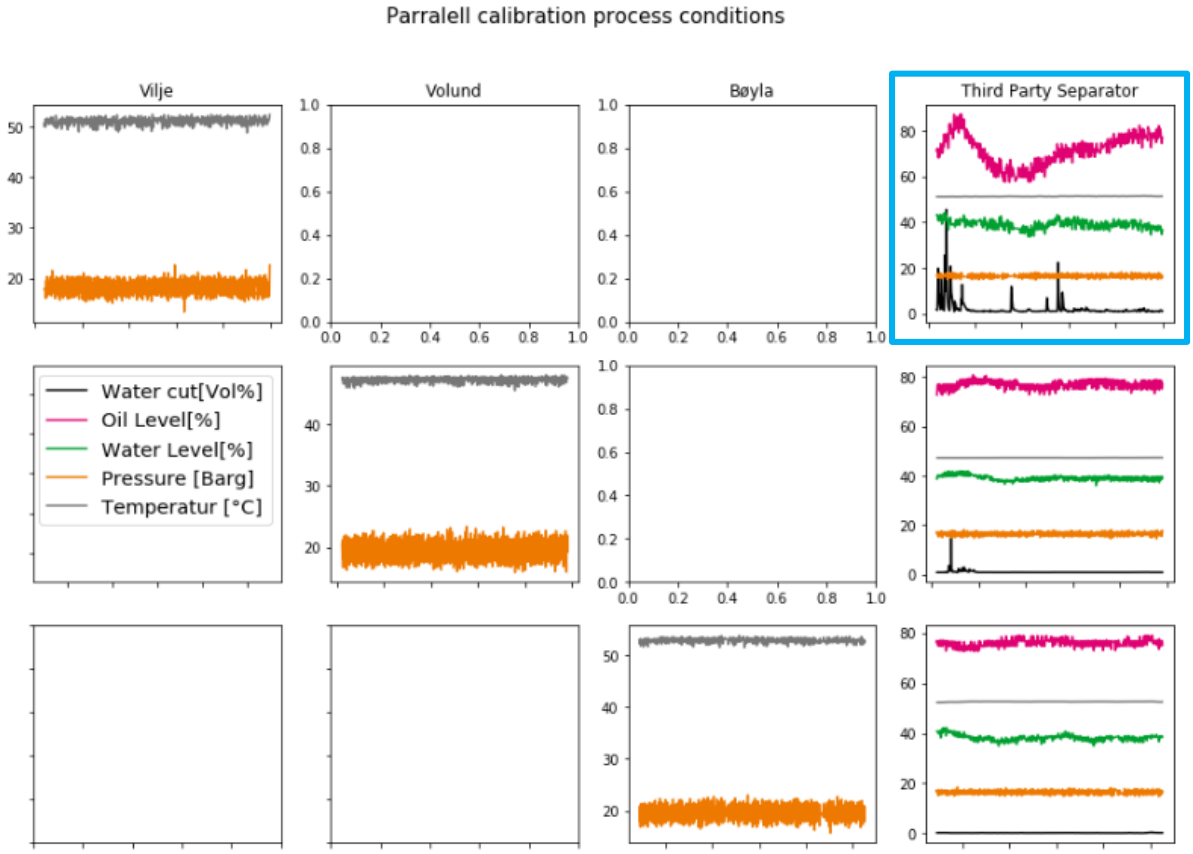


Figure 9 - Intensive variables of data set

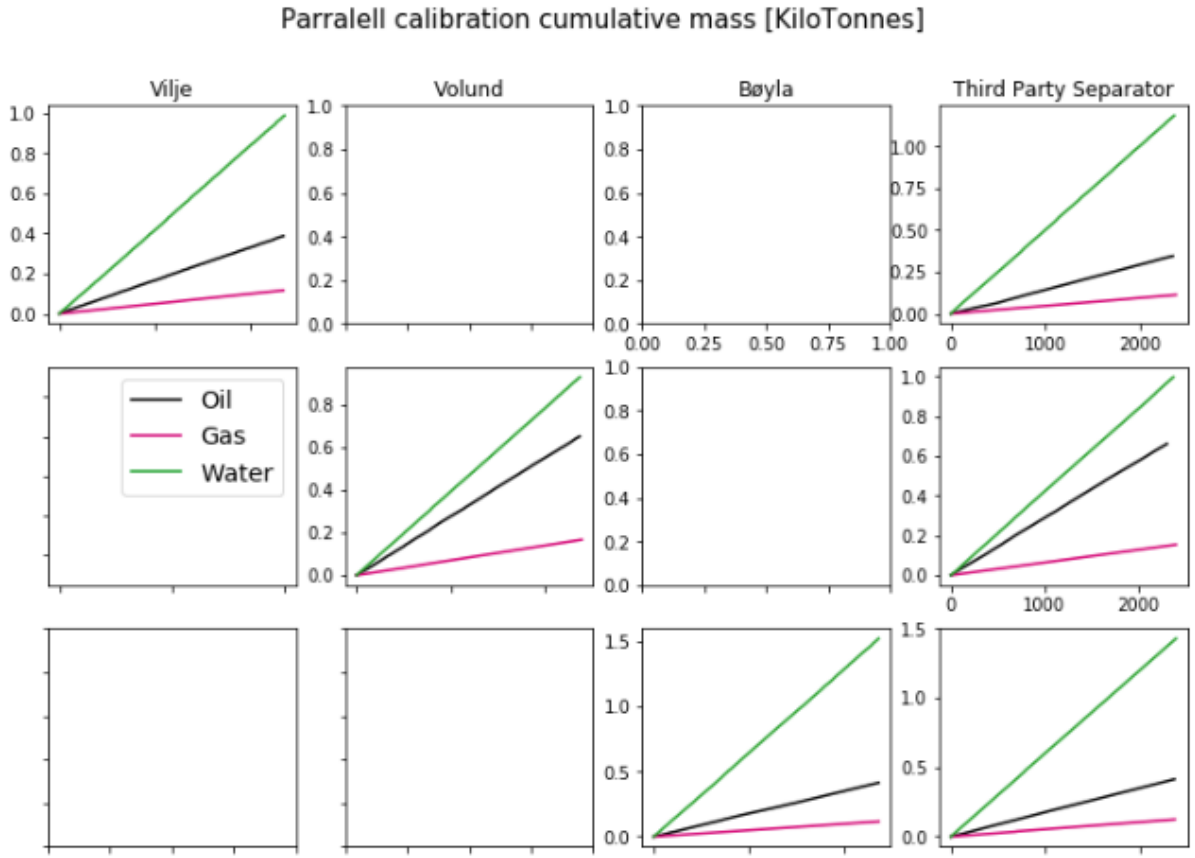


Figure 10 - Cumulative matrix plot

2.2.2 Statistics

Figure 11 shows the resulting values for each stream and each phase based on the data basis shown in Figure 12.

```
In [38]:  1  TradCal.StatsDF
```

Out[38]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Vilje	Oil	0.862017	0.144928	0.009687	1596.0
	Gas	1.015129	0.406133	0.003457	1210.0
	Water	1.197559	0.588430	0.002386	973.0
Volund	Oil	1.151804	0.496146	0.002830	1040.0
	Gas	0.949872	2.077198	0.000676	1000.0
	Water	0.985202	1.700435	0.000826	620.0
Bøyla	Oil	1.591782	0.145775	0.009630	1075.0
	Gas	0.979099	3.859319	0.000364	270.0
	Water	1.098259	1.210052	0.001160	1000.0

Figure 11 - Resulting data frame of calibration

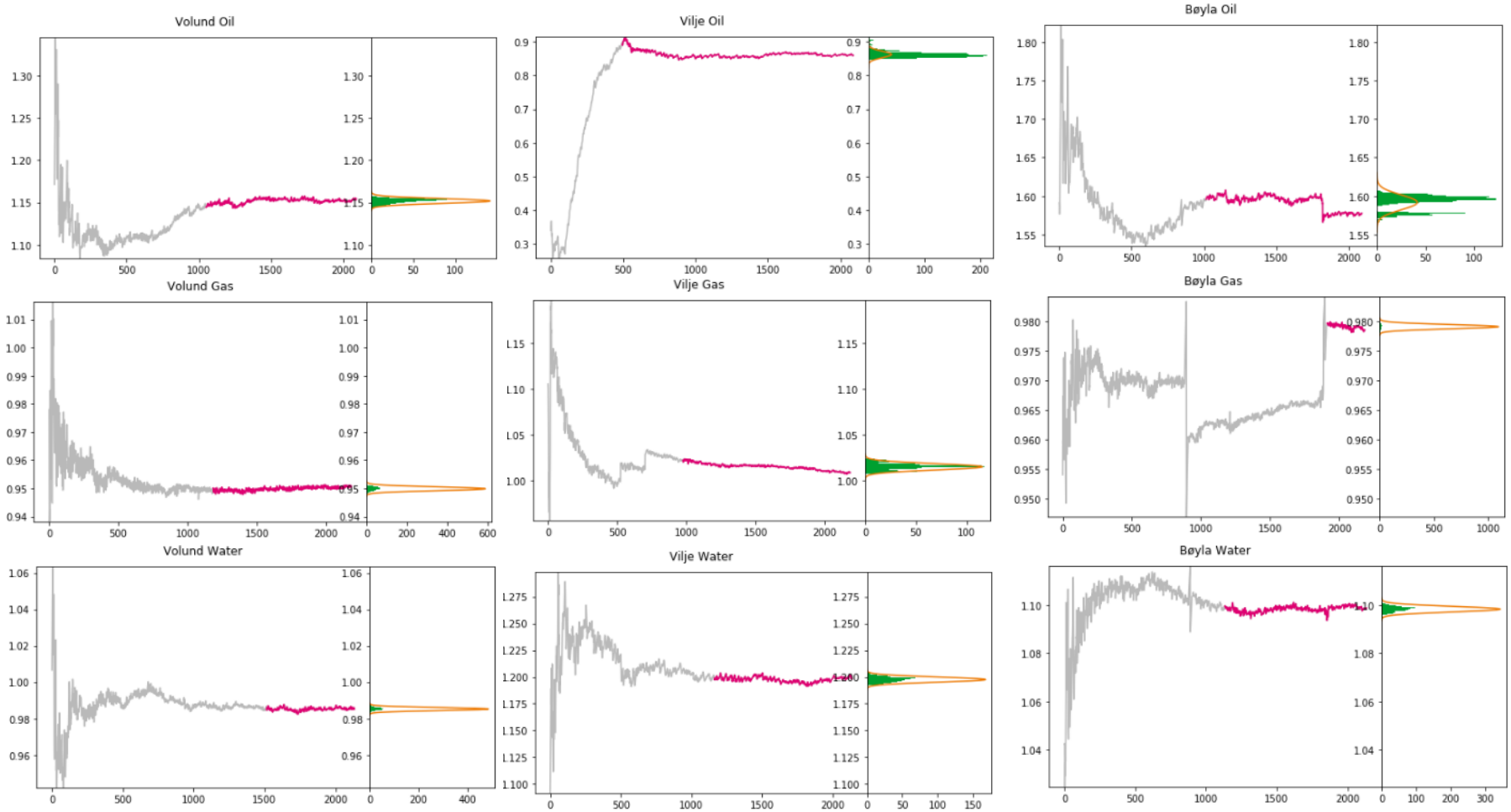


Figure 12 - Statistical basis of Traditional calibration

2.3 Comparison

The result of both calibration compared towards each other for each stream is shown in Figure 13 for Volund, Figure 14 for Vilje and Figure 15 for Bøyla multi-phase flow meters. Just form these figures the result of both methods seam to replicate each other very well and that the parallel calibration method performs satisfactorily. But the data set for Traditional and synthetic parallel calibration has one weakness which is the Vilje trial, which is not a type of trial wanted for a parallel calibration.

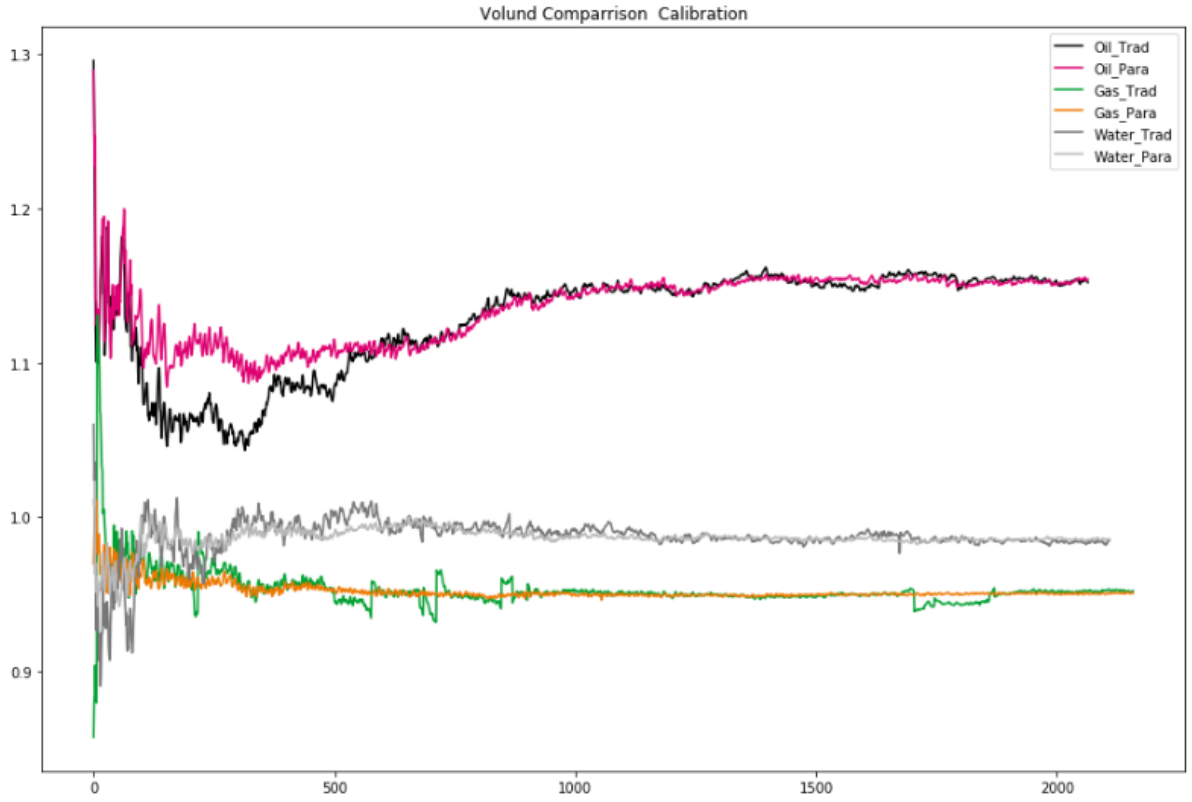


Figure 13 - Volund calibration comparison

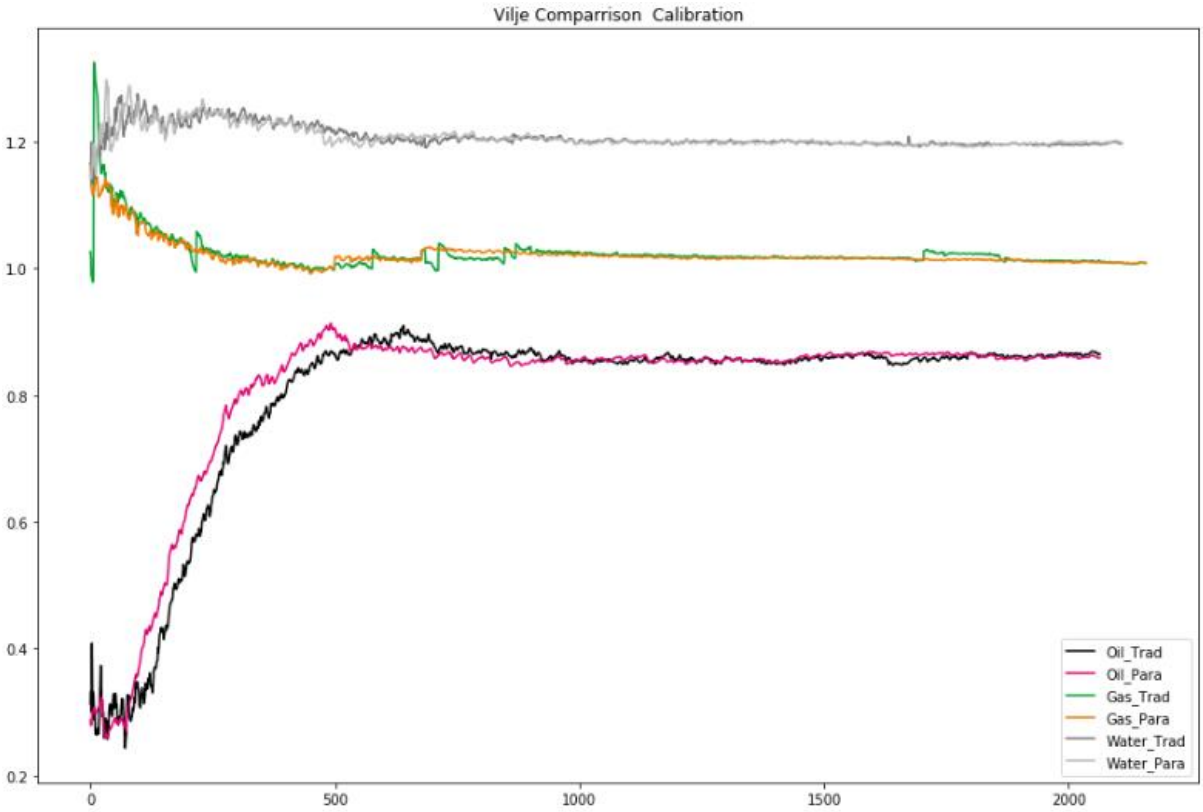


Figure 14- Vilje calibration comparison

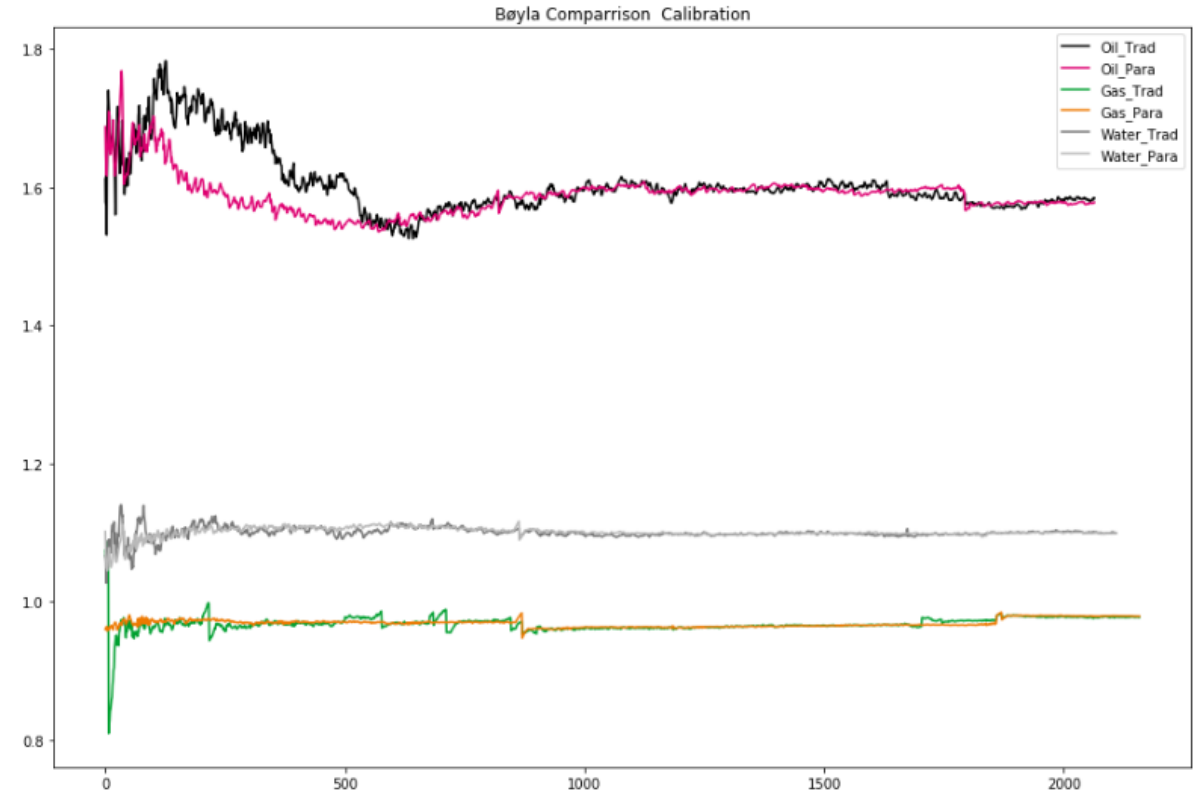


Figure 15 - Bøyla calibration comparison

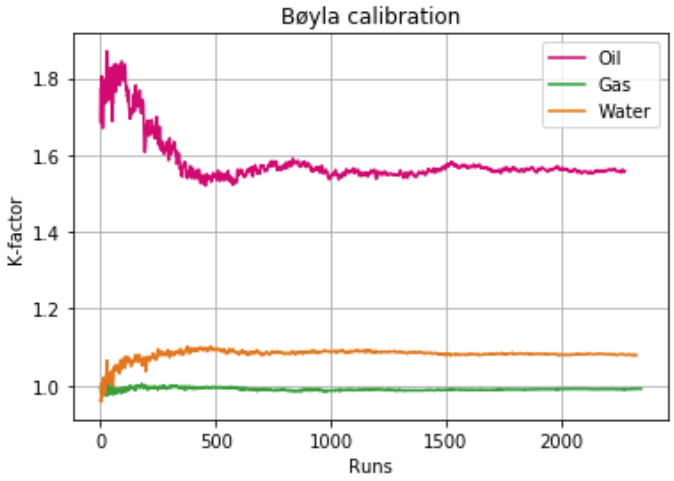
3 2x2x2 parallel calibration

A 2x2x2 calibration was performed on Alvheim Figure 16 shows the code for the buildup of trials, import libraries, and the time windows and duration, and the trial configuration for this calibration. Figure 17 shows the resulting k-factor development across the runs. Figure 18 show the cumulative values fed into the calibration method. Figure 19 show the mass flowrates for each stream and trial. Figure 20 shows the process conditions. Figure 21 depicts the statistical basis of the different k-factor developments and plots a histogram of the sample distribution together with the probability density plot of a normal distribution based on the statistical basis. Figure 22 show the resulting values in a pandas dataframe, and Figure 23 shows a violin plot of the resulting statistical basis for each stream and phase.

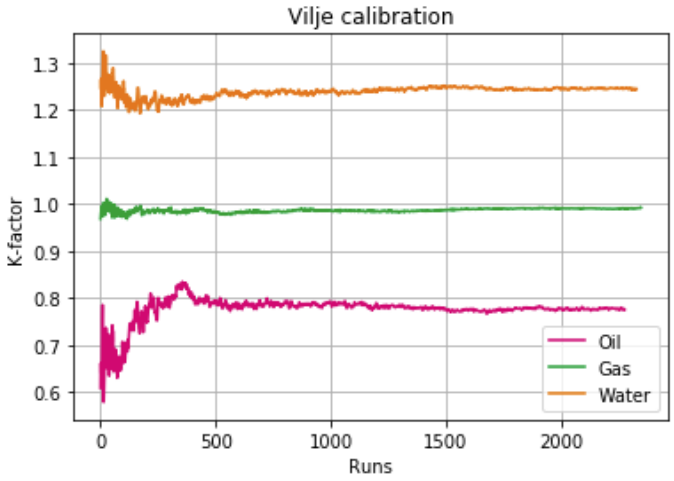
```
In [1]: %%javascript
        IPython.OutputArea.auto_scroll_threshold = 9999;

In [2]: import numpy as np
        import pandas as pd
        import matplotlib as plt

In [3]: from datetime import datetime
        %matplotlib inline

In [4]: from cognite import CogniteClient
        cdp = CogniteClient()

In [5]: from ParralellCalibration import *
        from TimeWindowLocator import *

In [6]: #Locate time windows for Parralell calibration
        start = datetime(2019, 3, 30)
        end = datetime(2019, 4, 3)
        ViVoWindow,BøVoWindow,BøViWindow = FindTrialWindows(start,end,cdp)

In [7]: ViVoWindow

Out[7]: (datetime.datetime(2019, 3, 31, 3, 28, 30),
         datetime.datetime(2019, 3, 31, 12, 13))

In [8]: BøVoWindow

Out[8]: (datetime.datetime(2019, 3, 30, 15, 4, 30),
         datetime.datetime(2019, 3, 31, 1, 6))

In [9]: BøViWindow

Out[9]: (datetime.datetime(2019, 3, 31, 14, 36), datetime.datetime(2019, 4, 1, 0, 35))

In [10]: Duration = timedelta(minutes=400)

In [11]: Trial1 = CreateTrial(["Vilje","Volund"],(ViVoWindow[0],ViVoWindow[0]+Duration),cdp)

         Creating Vilje Stream
         Creating Volund Stream
         Creating Separator Streams
         Trial Finnished

In [12]: Trial2 = CreateTrial(["Bøyla","Volund"],(BøVoWindow[0],BøVoWindow[0]+Duration),cdp)

         Creating Bøyla Stream
         Creating Volund Stream
         Creating Separator Streams
         Trial Finnished

In [13]: Trial3 = CreateTrial(["Bøyla","Vilje"],(BøViWindow[0],BøViWindow[0]+Duration),cdp)

         Creating Bøyla Stream
         Creating Vilje Stream
         Creating Separator Streams
         Trial Finnished

In [14]: Calibration = ParralellCalibration([Trial3,Trial2,Trial1])
```

Figure 16 - Execution of calibration

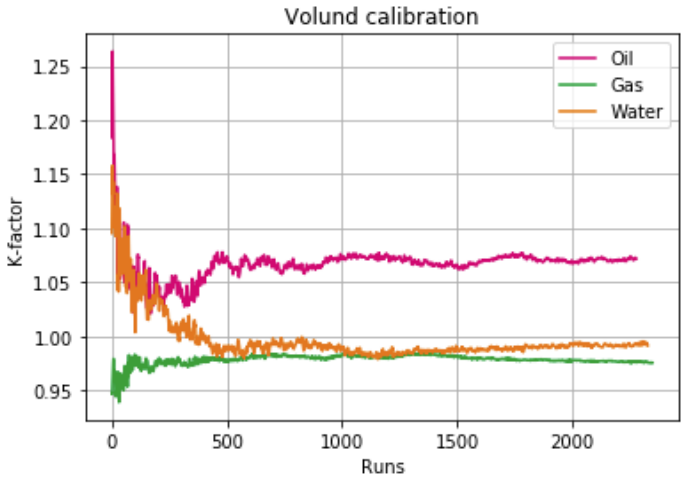
```
In [15]: Calibration.Plot()
```




```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Figure 17 - k-factor development as a result of the calibration

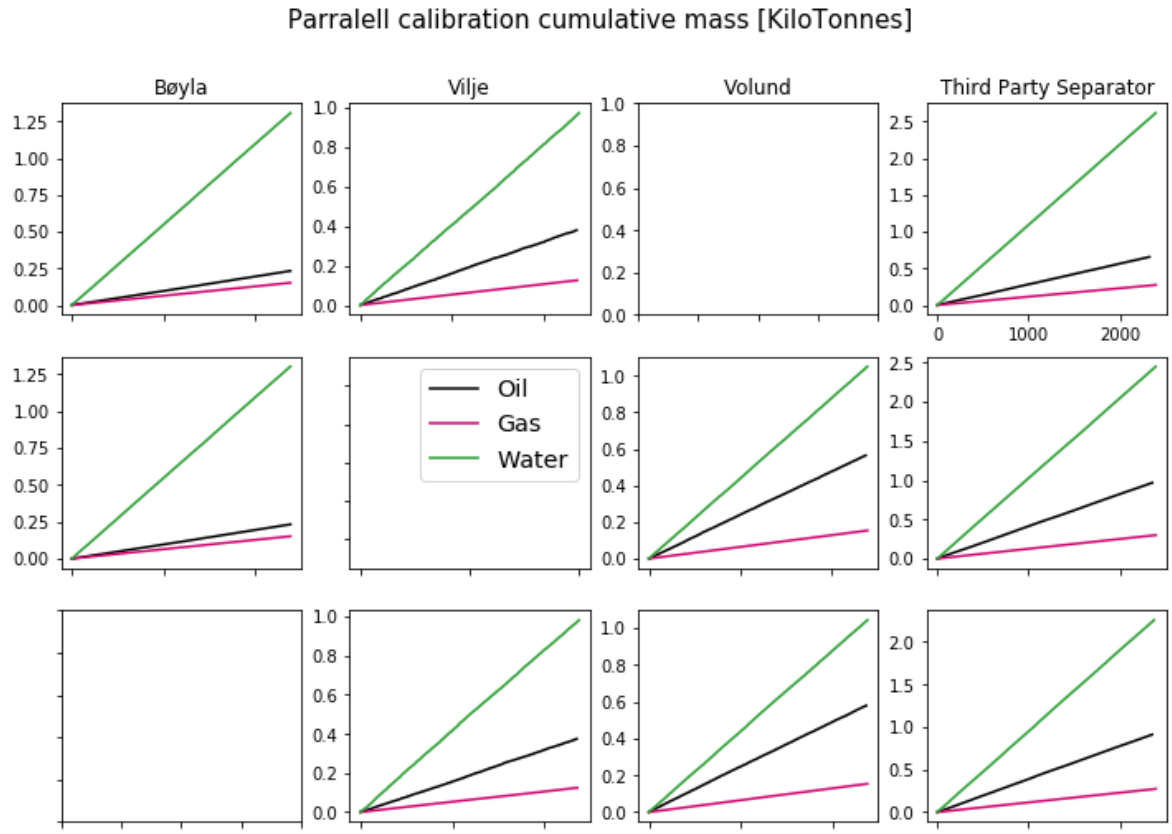


Figure 18 - cumulative values across the trials in augmente matrix plotting

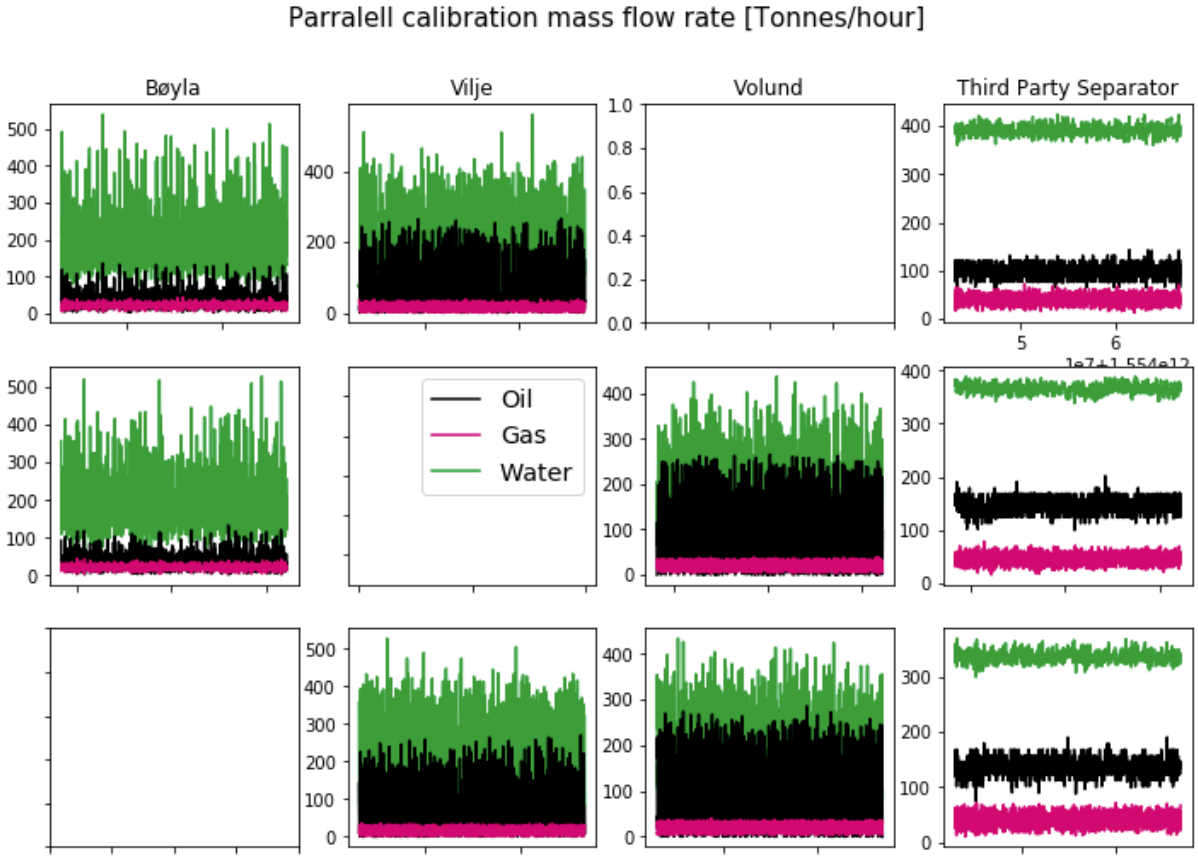


Figure 19 - mass flowrates across the trials in augmented matrix plotting

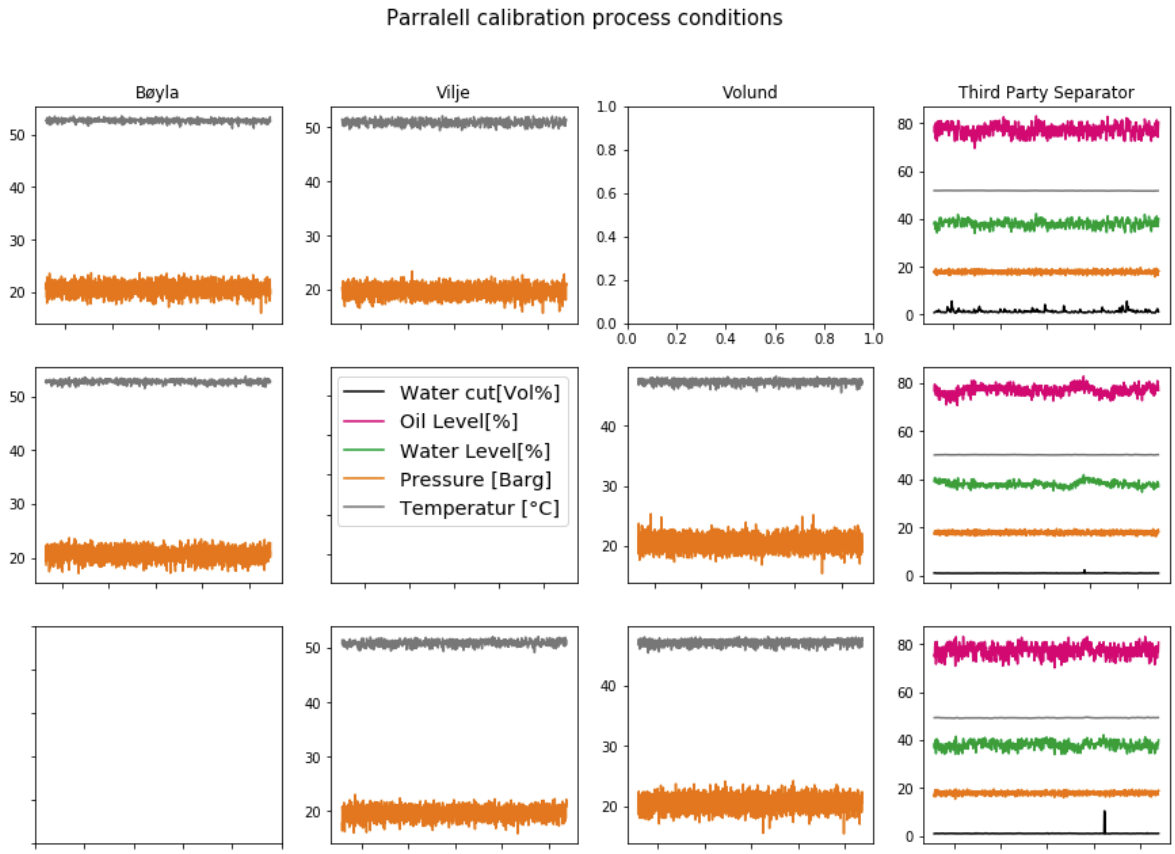


Figure 20 - Process conditions during trials

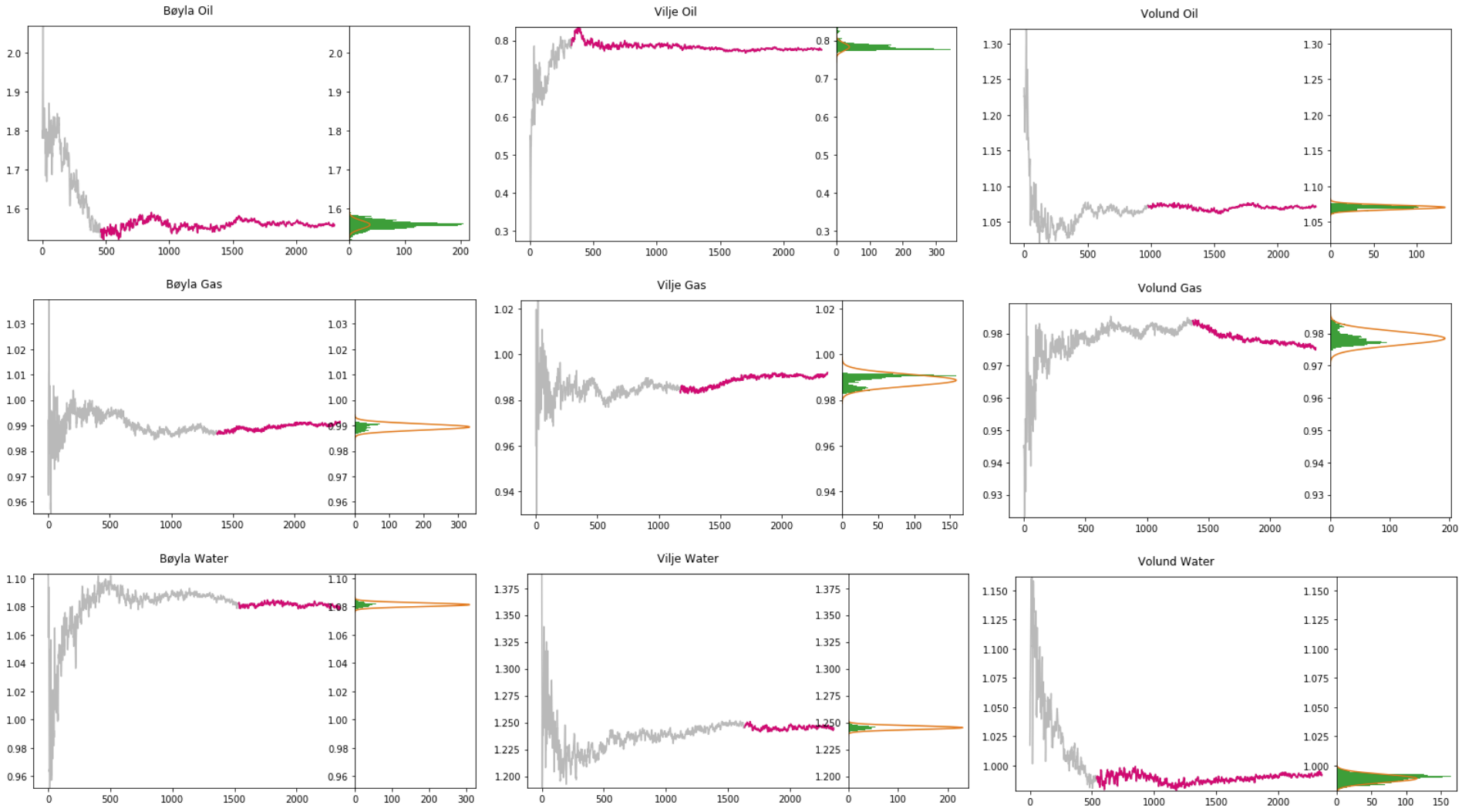


Figure 21 - Statistical basis and distribution of calibration, k-factor development and basis in left plot and vertical histogram of sample numbers and probability density of a normal distribution.

```
In [28]: Calibration.StatsDF
```

Out[28]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Bøyla	Oil	1.559129	0.132458	0.010599	1841.0
	Gas	0.989396	1.171835	0.001198	1000.0
	Water	1.081198	1.086397	0.001292	817.0
Vilje	Oil	0.782720	0.140507	0.009991	1974.0
	Gas	0.988635	0.557248	0.002519	1200.0
	Water	1.245149	0.807568	0.001738	717.0
Volund	Oil	1.070587	0.465426	0.003016	1323.0
	Gas	0.978454	0.677131	0.002073	1000.0
	Water	0.988960	0.405762	0.003460	1816.0

Figure 22 - result data frame of calibration

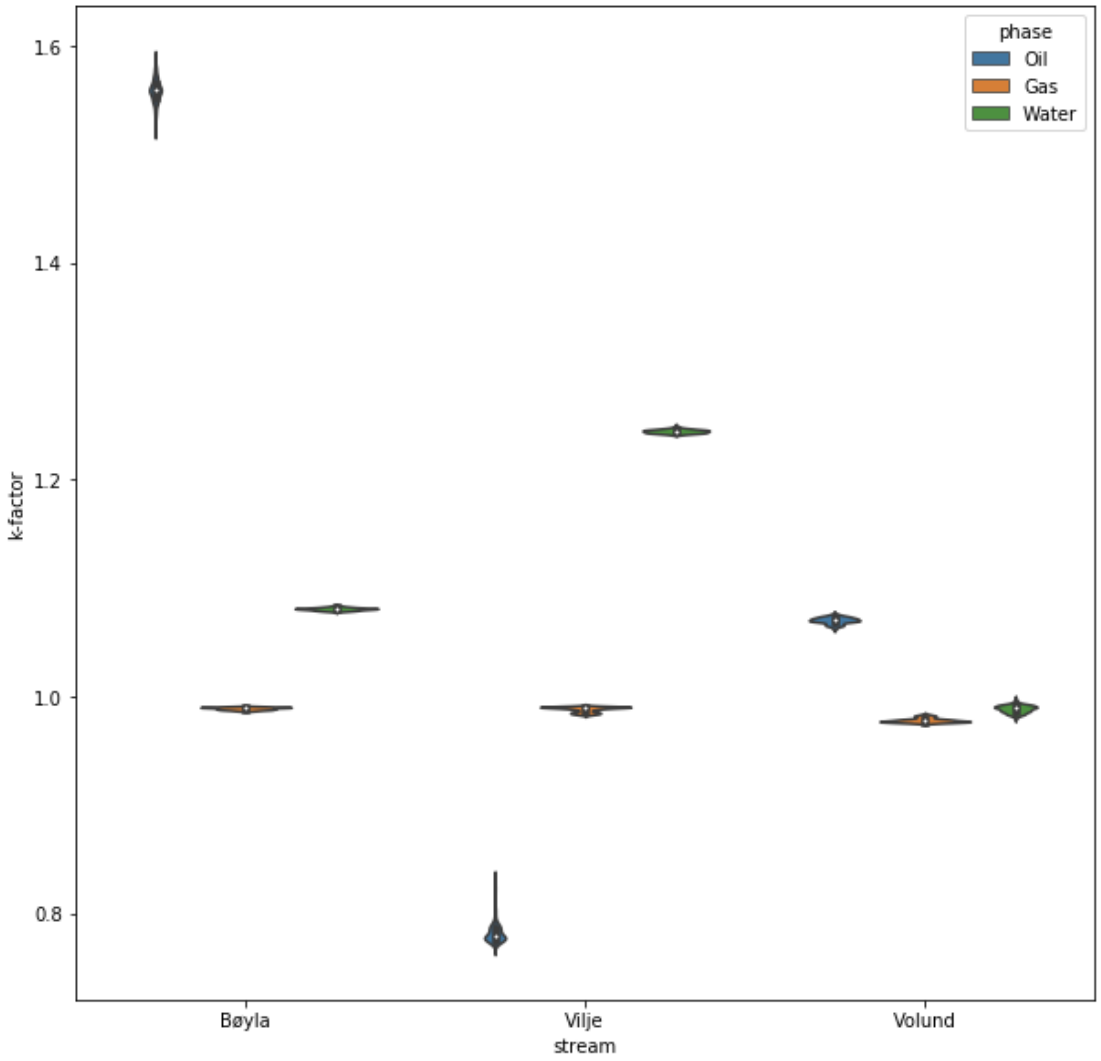


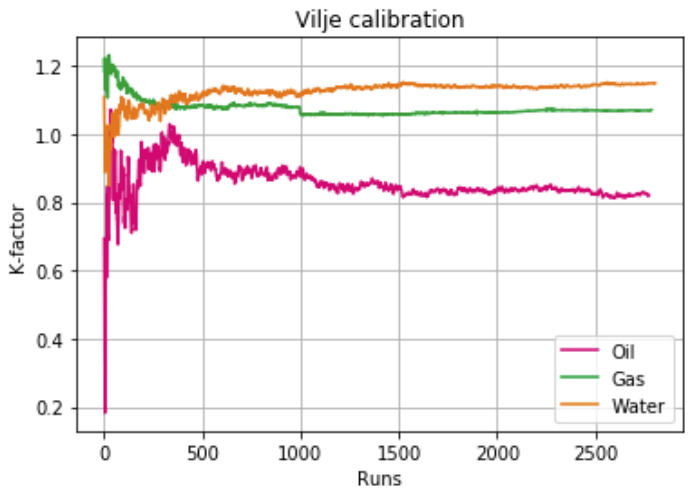
Figure 23 - Violin plot of the resulting basis of the calibration

4 3x2x2 parallel calibration

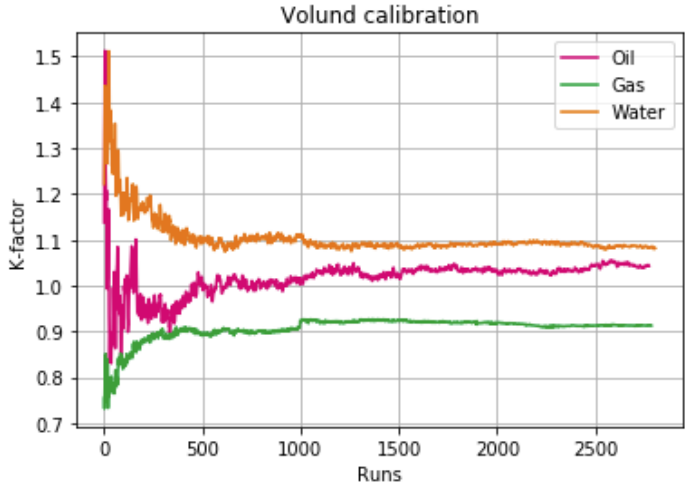
A 2x2x2 calibration was performed on Alvheim Figure 24 shows the code for the buildup of trials, import libraries, and the time windows and duration, and the trial configuration for this calibration. Figure 25 shows the resulting k-factor development across the runs. Figure 26 show the cumulative values fed into the calibration method. Figure 27 show the mass flowrates for each stream and trial. Figure 28 shows the process conditions. Figure 29 depicts the statistical basis of the different k-factor developments and plots a histogram of the sample distribution together with the probability density plot of a normal distribution based on the statistical basis. Figure 31 show the resulting values in a pandas dataframe, and Figure 30 shows a violin plot of the resulting statistical basis for each stream and phase.

```
In [1]: %%javascript
        IPython.OutputArea.auto_scroll_threshold = 9999;

In [2]: import numpy as np
        import pandas as pd
        import matplotlib as plt

In [3]: from datetime import datetime
        %matplotlib inline

In [4]: from cognite import CogniteClient
        cdp = CogniteClient()

In [5]: from ParralellCalibration import *
        from TimeWindowLocator import *

In [6]: #Locate time windows for Parralell calibration
        start = datetime(2019, 3, 30)
        end = datetime(2019, 4, 3)
        ViVoWindow,BøVoWindow,BøViWindow = FindTrialWindows(start,end,cdp)

In [7]: Duration = timedelta(minutes=int(8*60)) #8 hour duration of trials

In [8]: TrippelWindow = [datetime(2019, 4, 3,1),datetime(2019, 4, 3,1)+Duration]

In [9]: Trial1 = CreateTrial(["Vilje","Volund","Bøyla"],TrippelWindow,cdp)

        Creating Vilje Stream
        Creating Volund Stream
        Creating Bøyla Stream
        Creating Separator Streams
        Trial Finnished

In [10]: Trial2 = CreateTrial(["Volund","Bøyla"],(BøVoWindow[0],BøVoWindow[0]+Duration),cdp)

         Creating Volund Stream
         Creating Bøyla Stream
         Creating Separator Streams
         Trial Finnished

In [11]: Trial3 = CreateTrial(["Vilje","Volund"],(ViVoWindow[0],ViVoWindow[0]+Duration),cdp)

         Creating Vilje Stream
         Creating Volund Stream
         Creating Separator Streams
         Trial Finnished

In [12]: Calibration = ParralellCalibration([Trial1,Trial3,Trial2])
```

Figure 24 - execution of calibration



<Figure size 432x288 with 0 Axes>



<Figure size 432x288 with 0 Axes>

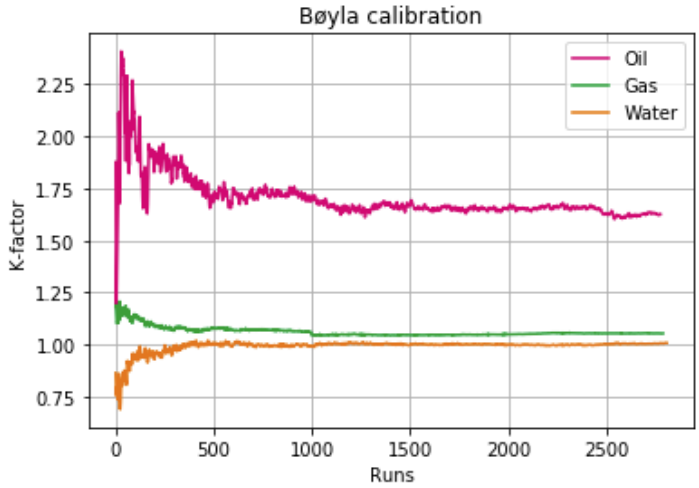


Figure 25 - Resulting k-factor development of the algorithm

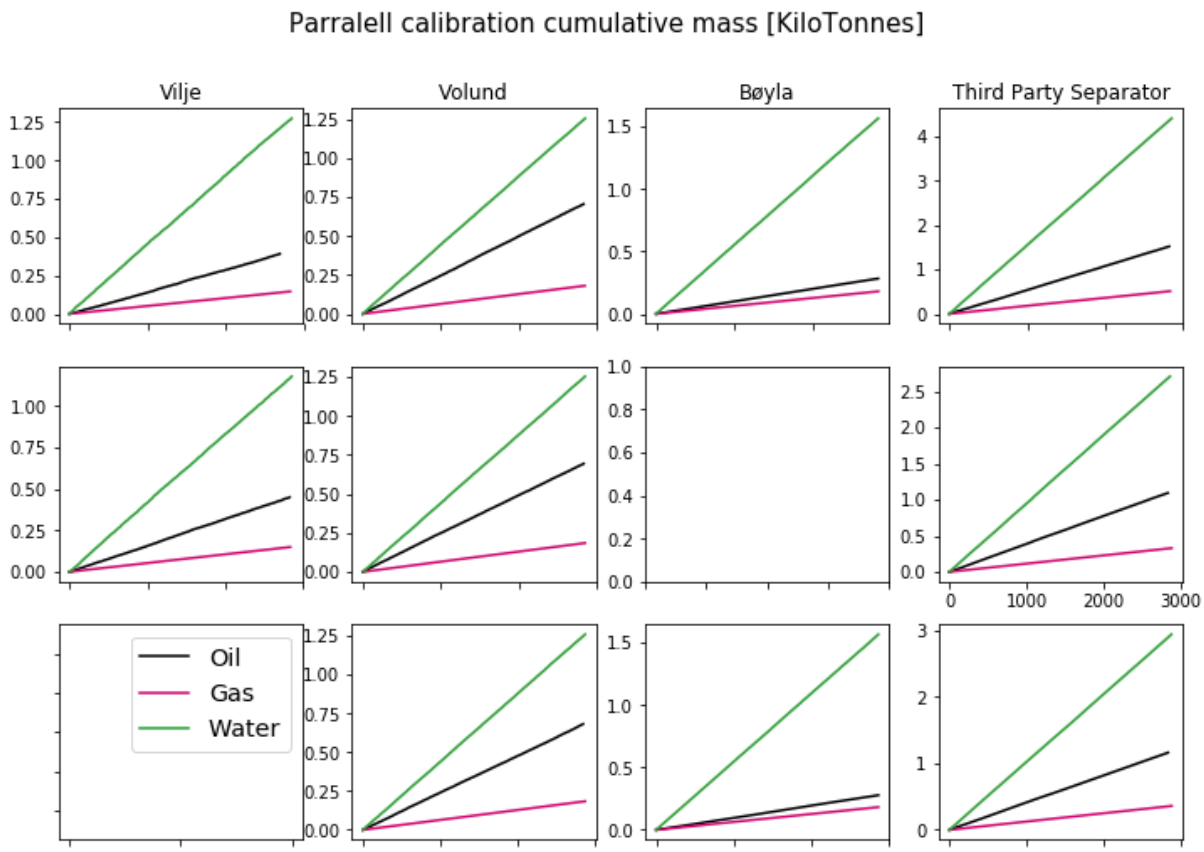


Figure 26 - Cumulative values entering the calibration method

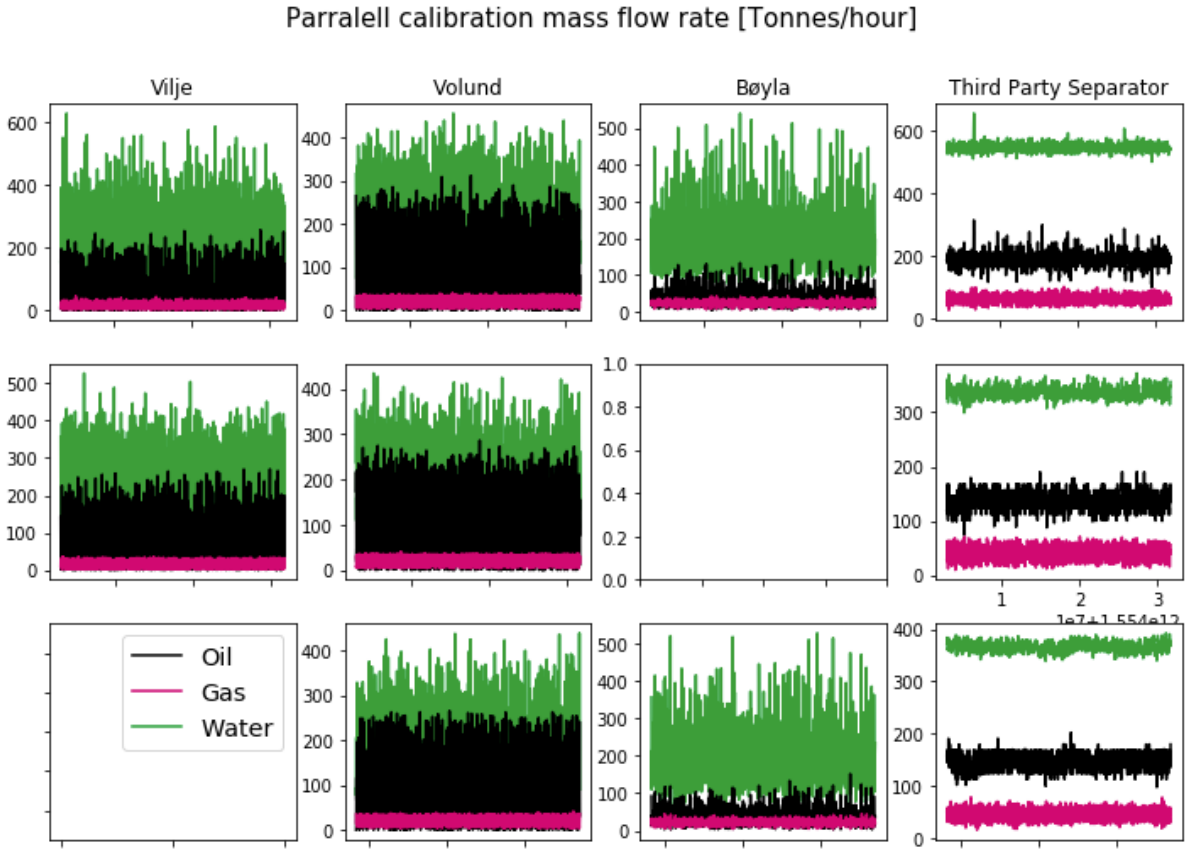


Figure 27 - mass flowrates of streams and trials

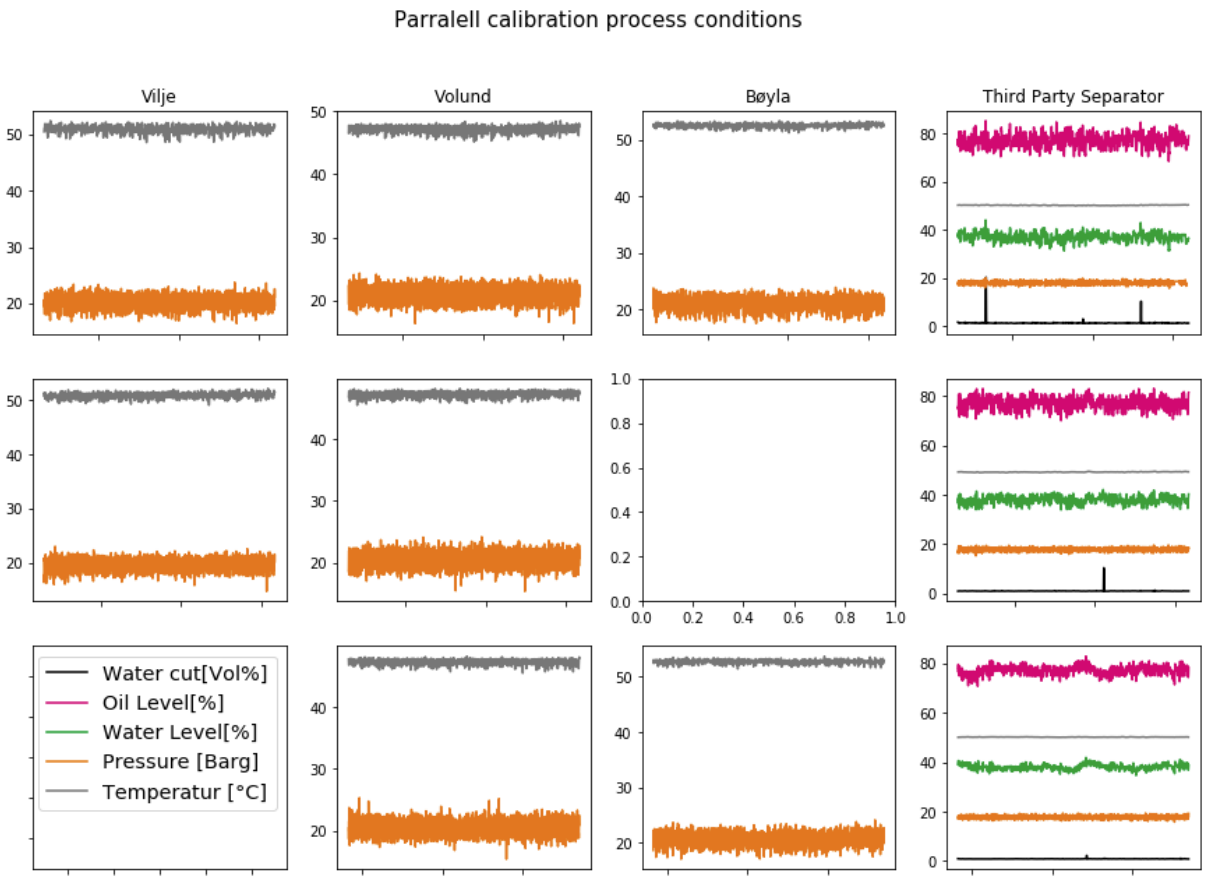


Figure 28 - Process condition during trials

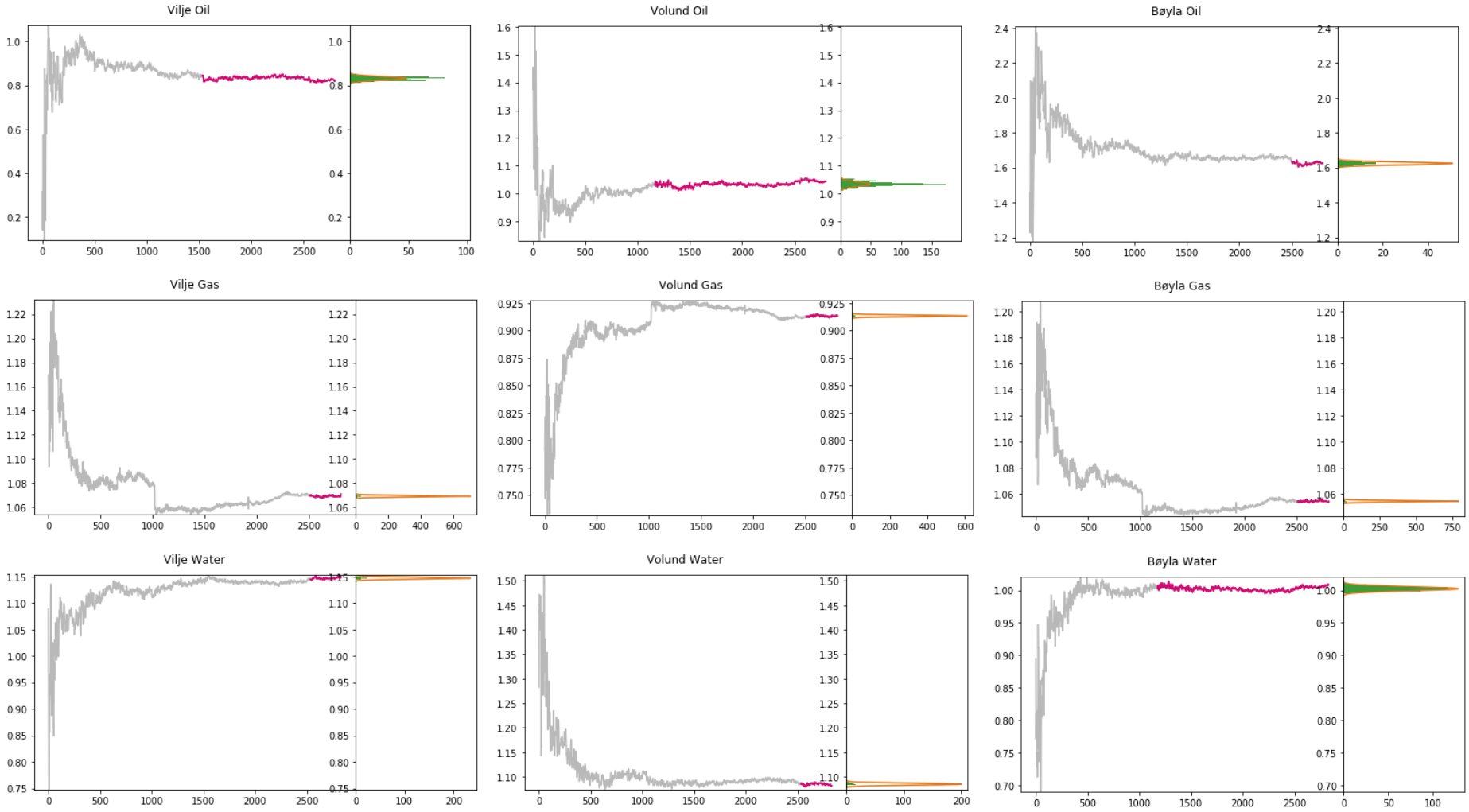


Figure 29 - k-factor development with statistical basis, with complementary histogram and normal distributions probability density function based on statistical basis

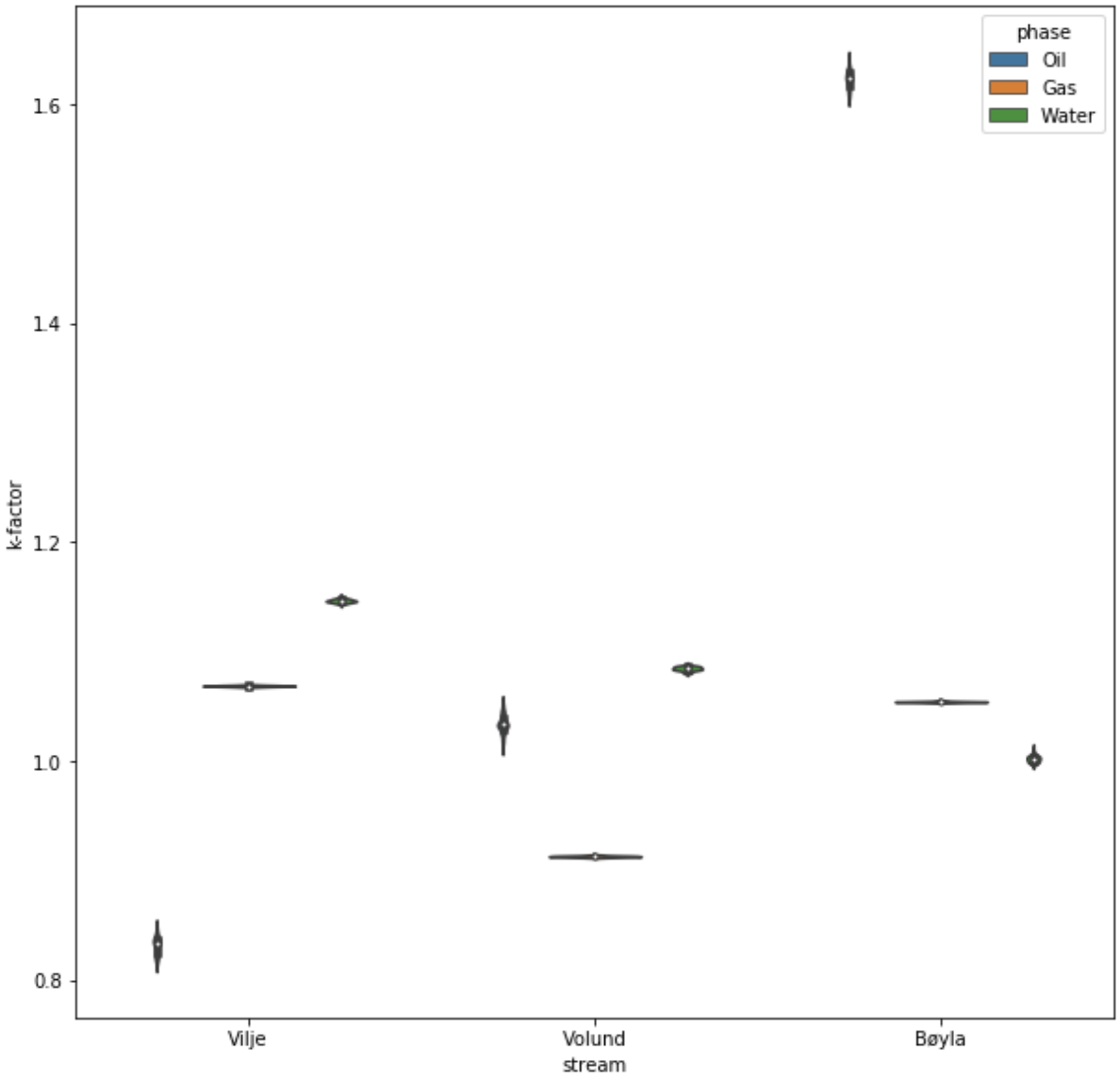


Figure 30 - Violin plot of resulting statistical basis

```
In [41]: Calibration.StatsDF
```

Out[41]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Vilje	Oil	0.831449	0.170941	0.008213	1269.0
	Gas	1.069077	2.476105	0.000567	300.0
	Water	1.146770	0.826159	0.001699	300.0
Volund	Oil	1.034110	0.172486	0.008139	1635.0
	Gas	0.913304	2.155170	0.000651	300.0
	Water	1.084777	0.705037	0.001991	300.0
Bøyla	Oil	1.623175	0.179520	0.007820	300.0
	Gas	1.054495	2.794607	0.000502	300.0
	Water	1.002312	0.452364	0.003103	1654.0

Figure 31 - pandas dataframe of resulting numerical values

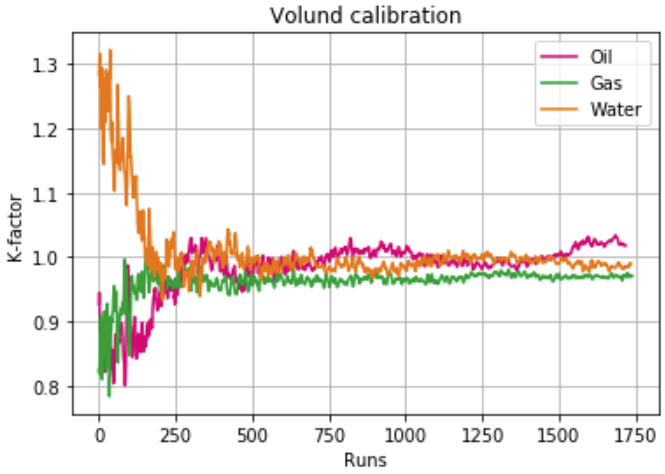
5 One month later 3x2x2

In the last days of April 2019 a new parallel calibration sequence was executed on Alvheim giving the following results, and is shown in the figures below. Figure 32 shows the execution and configuration of calibration. Figure 33 shows the resulting k-factor development, Figure 34 shows the chosen statistical basis for the calibration. Figure 35 shows the mass flowrates for the calibration. Figure 36 shows the process conditions during the calibration. Figure 37 shows the resulting calibration in a pandas dataframe. Figure 38 shows a violin plot of the result.

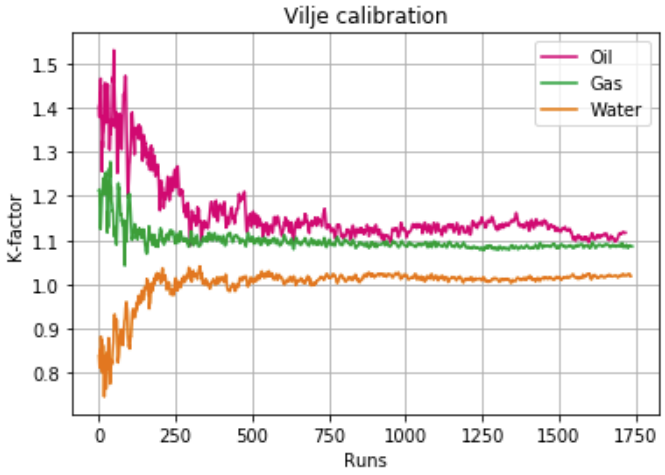
```
In [1]: %%javascript
        IPython.OutputArea.auto_scroll_threshold = 9999;

In [2]: import numpy as np
        import pandas as pd
        from datetime import datetime
        from datetime import timedelta
        from matplotlib import pyplot as plt
        %matplotlib inline

In [3]: from cognite import CogniteClient
        cdp = CogniteClient()

In [4]: from ParralellCalibration import *
        from TimeWindowLocator import *

In [5]: #Locate time windows for Parralell calibration
        start = datetime(2019, 4, 28)
        end = datetime(2019, 5, 1)
        ViVoWindow,BøVoWindow,BøViWindow = FindTrialWindows(start,end,cdp)

        No Bøyla Vilje Window found

In [6]: TrippelWindow = [datetime(2019, 5, 1,0),datetime(2019, 5, 1,10)]

In [7]: Duration = timedelta(minutes=5*60)

In [8]: Trial1 = CreateTrial(["Volund","Vilje","Bøyla"],(TrippelWindow[0],TrippelWindow[0]+Duration),cdp)

        Creating Volund Stream
        Creating Vilje Stream
        Creating Bøyla Stream
        Creating Separator Streams
        Trial Finnished

In [9]: Trial2 = CreateTrial(["Volund","Bøyla"],(BøVoWindow[0],BøVoWindow[0]+Duration),cdp)

        Creating Volund Stream
        Creating Bøyla Stream
        Creating Separator Streams
        Trial Finnished

In [10]: Trial3 = CreateTrial(["Volund","Vilje"],(ViVoWindow[0],ViVoWindow[0]+Duration),cdp)

         Creating Volund Stream
         Creating Vilje Stream
         Creating Separator Streams
         Trial Finnished

In [11]: Calibration = ParralellCalibration([Trial1,Trial2,Trial3])
```

Figure 32 - Calibration execution late April

```
In [12]: Calibration.Plot()
```




```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

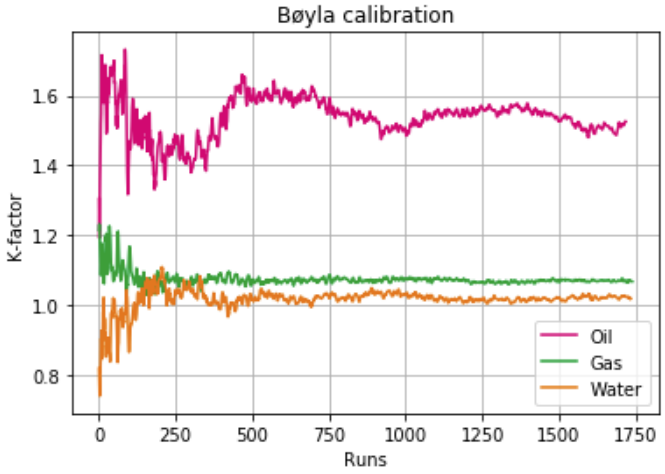


Figure 33 - k-factor development

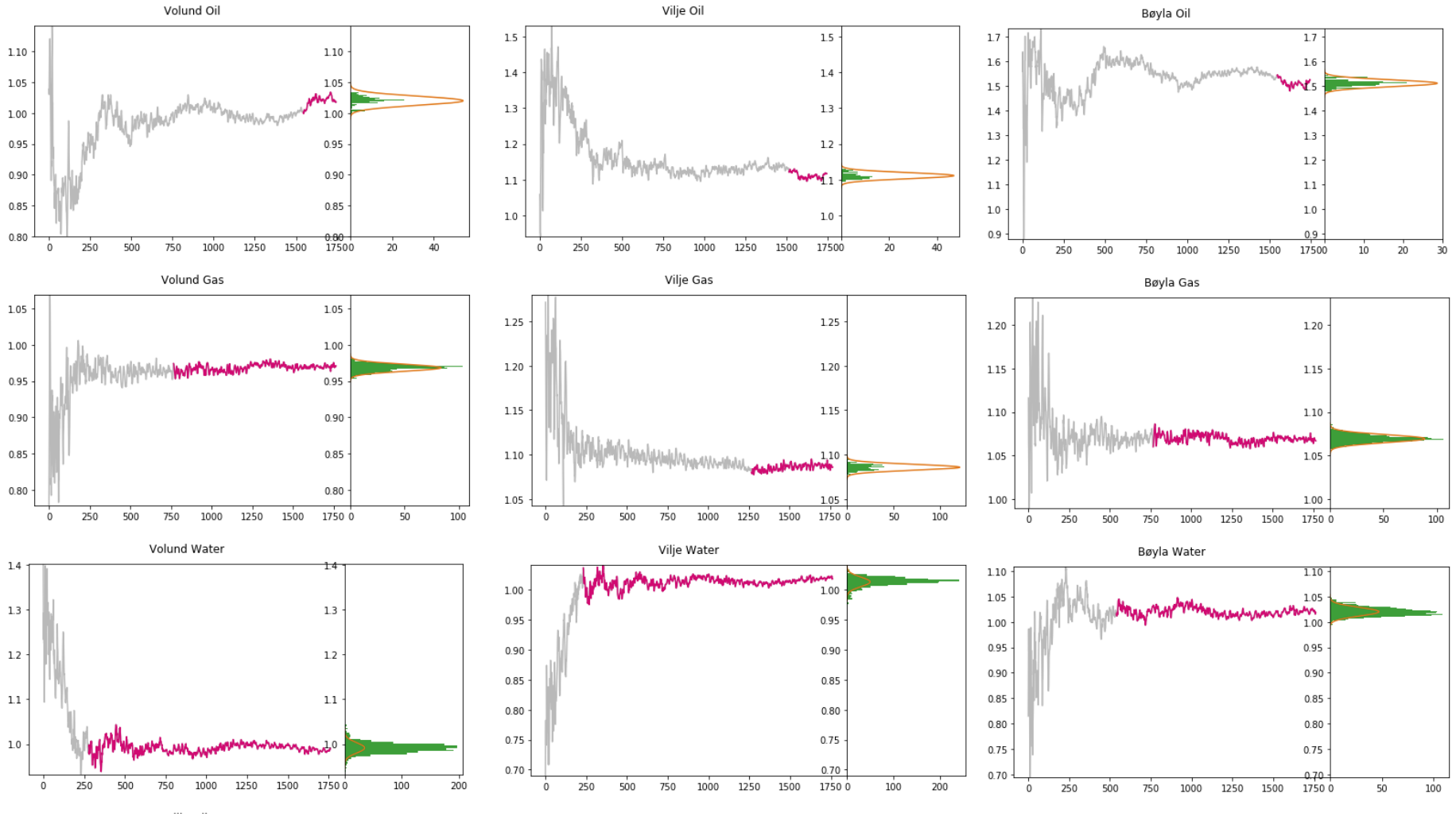


Figure 34 - Statistical basis of calibration one month later

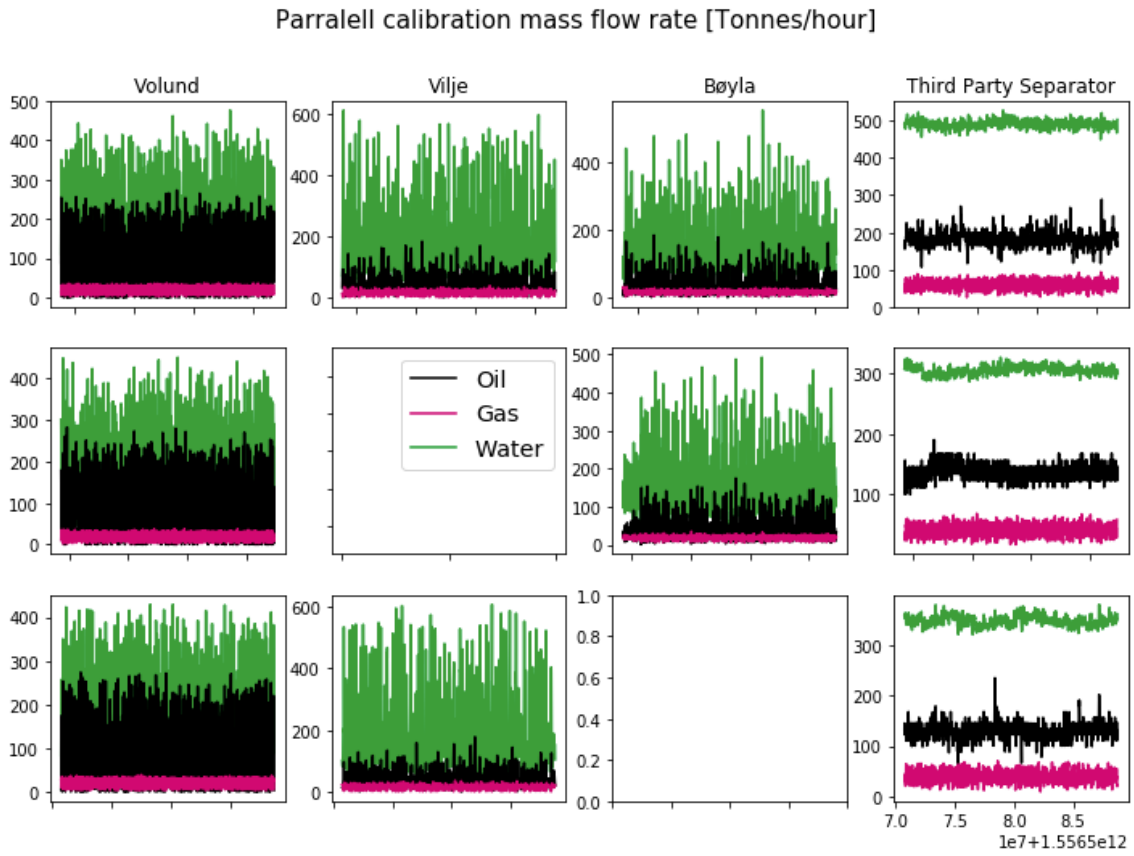


Figure 35 - mass flowrate during April May calibration

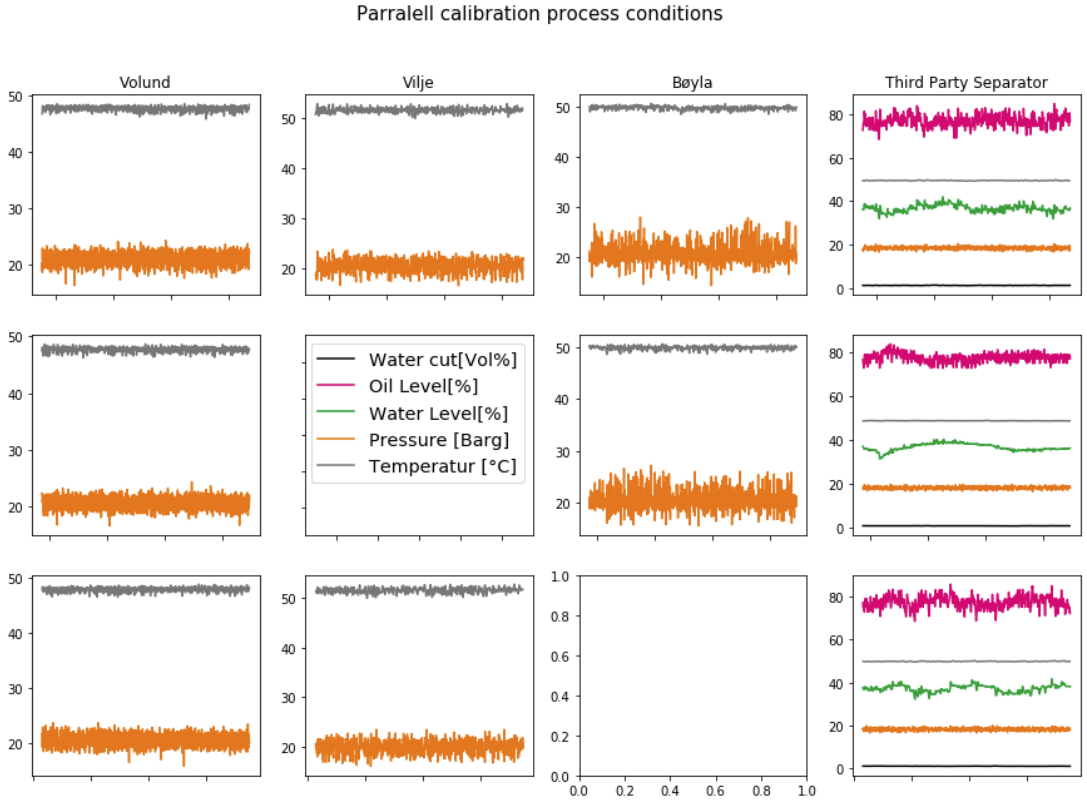


Figure 36 Process condition during april may calibration

```
In [47]: Calibration.CreateStatsDF()
         Calibration.StatsDF
```

Out[47]:

Stream	Phase	mean	uncert	St.dev	SampleSize
Volund	Oil	1.020043	0.190211	0.007381	200.0
	Gas	0.968242	0.294714	0.004764	1000.0
	Water	0.991756	0.122919	0.011421	1482.0
Vilje	Oil	1.111205	0.165291	0.008493	230.0
	Gas	1.085793	0.424791	0.003305	500.0
	Water	1.013514	0.176047	0.007974	1529.0
Bøyla	Oil	1.511291	0.101088	0.013888	200.0
	Gas	1.069523	0.313227	0.004482	1000.0
	Water	1.020290	0.166402	0.008437	1222.0

Figure 37 - resulting dataframe of calibration

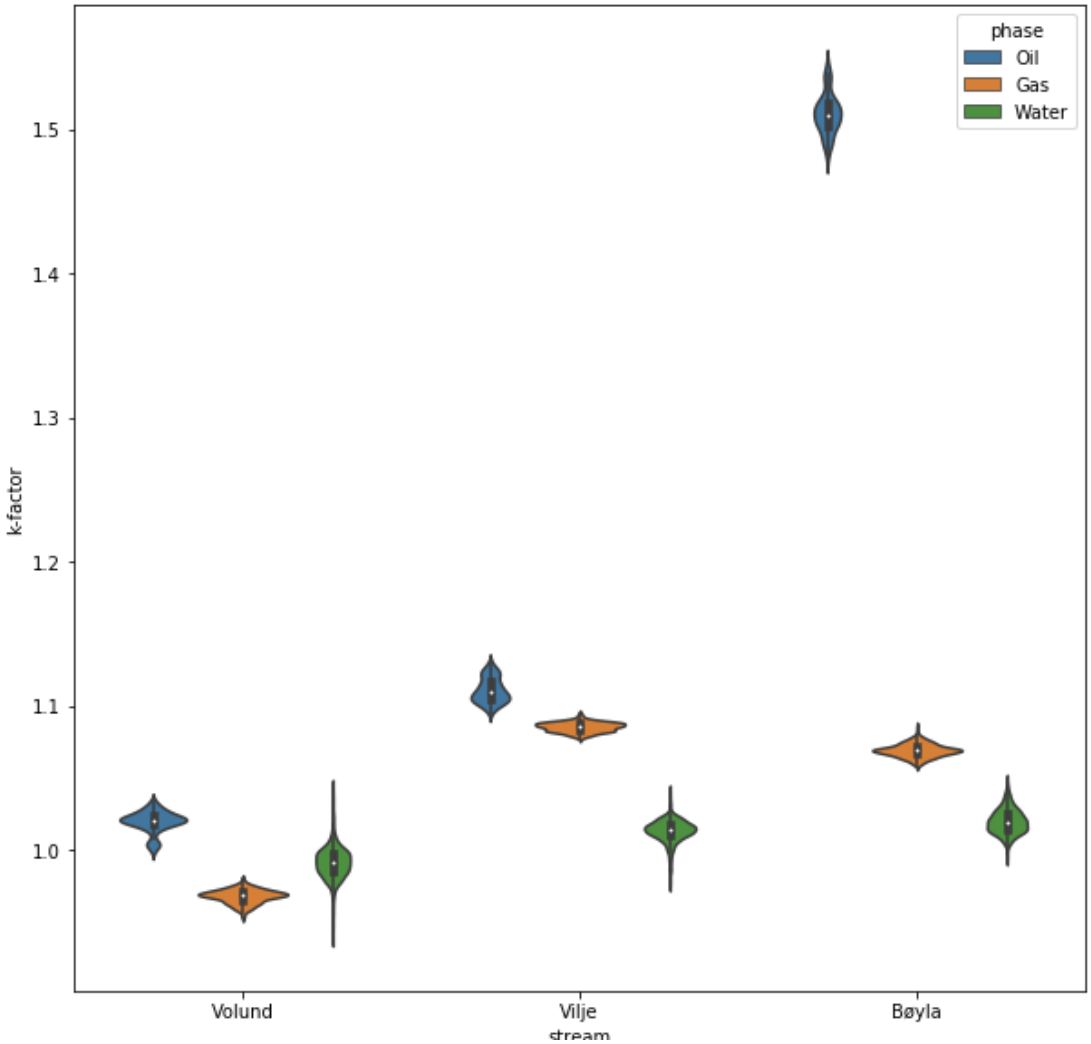


Figure 38 - violin plot of 3x2x2 calibration in April / May

6 Result

Combining the real result and the results from the algorithm execution as documented in this appendix the results combined together is shown in Table 6.1, and plotted in point plots for each of the streams in Figure 39 for Bøyla, Figure 40 for Vilje and Figure 41 for Volund

Table 6.1: Resulting values of actual, and the algorithms ,traditional, synthetic parallel and parallel calibration in March / April 2019

Calibration		Algorithm					
Time: March / April 2019		27/3 to 29/3			30/3 to 3/4		30/4 to 1/5
Stream	Phase	Current system	Traditional	Synthetic parallel	Parallel 2x2x2	Parallel 3x2x2	1mnd later 3x2x2
Bøyla	Oil	1.56405	1.59178	1.59117	1.55913	1.62318	1.51129
	Gas	0.90507	0.97910	0.97763	0.98940	1.05450	1.06952
	Water	1.07492	1.09826	1.09789	1.08120	1.00231	1.02029
Vilje	Oil	0.82050	0.86202	0.86305	0.78272	0.83145	1.11121
	Gas	0.97801	1.01513	1.01680	0.98864	1.06908	1.08579
	Water	1.13873	1.19756	1.19935	1.24515	1.14677	1.01351
Volund	Oil	1.14429	1.15180	1.15164	1.07059	1.03411	1.02004
	Gas	0.90674	0.94987	0.95168	0.97845	0.91330	0.96824
	Water	1.04478	0.98520	0.98450	0.98896	1.08478	0.99176

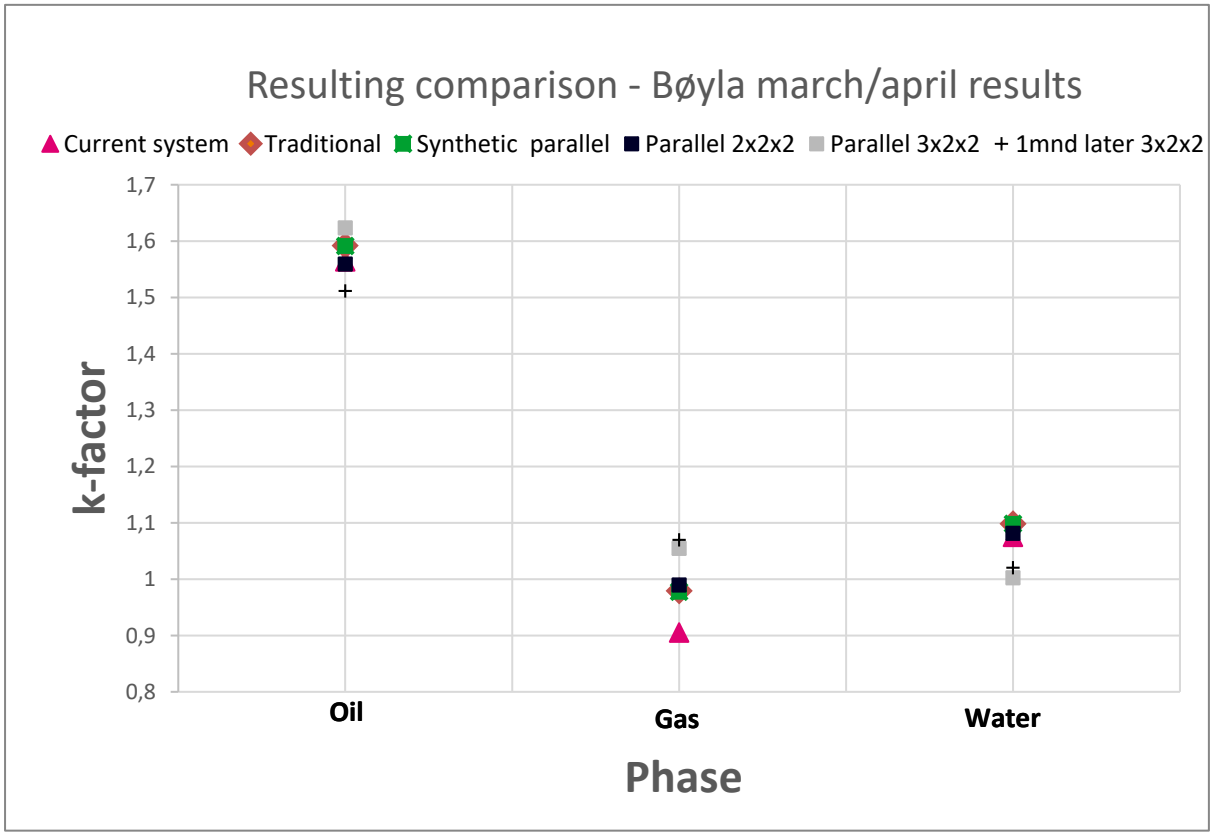


Figure 39 - Resulting comparison of Bøyla on March April results

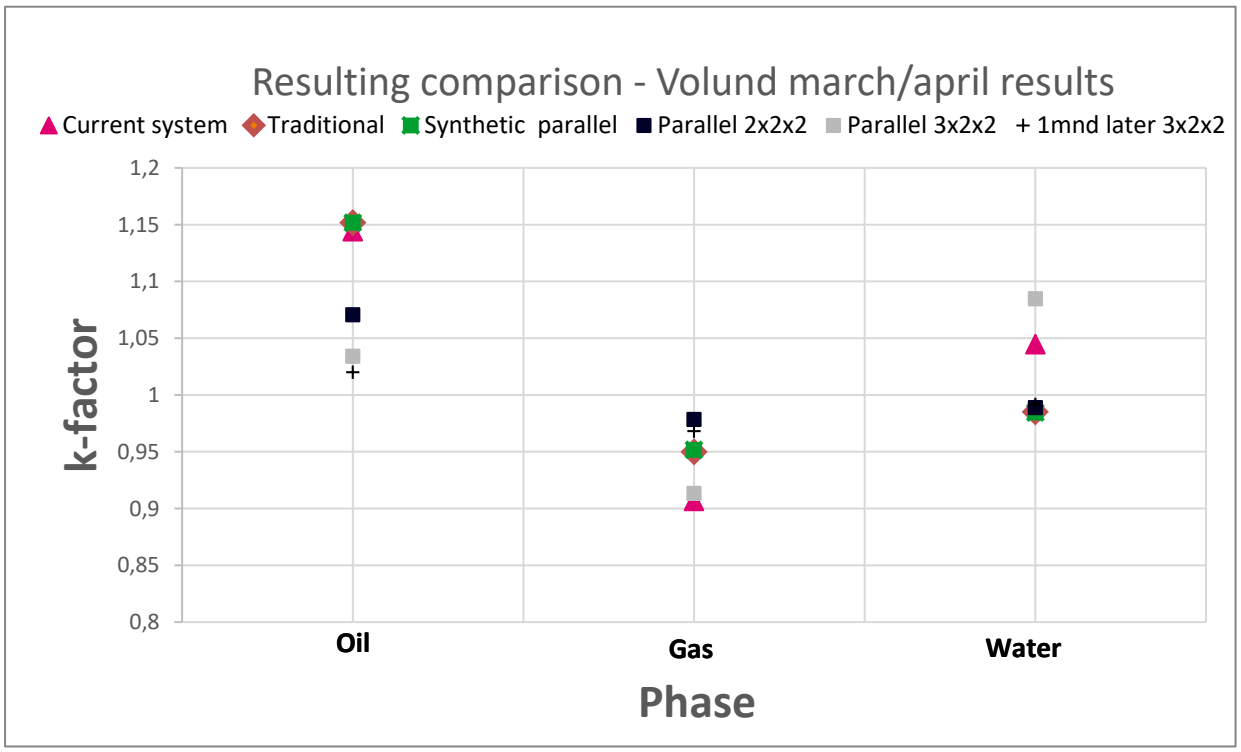


Figure 40 - Resulting comparison of Vilje on March April results

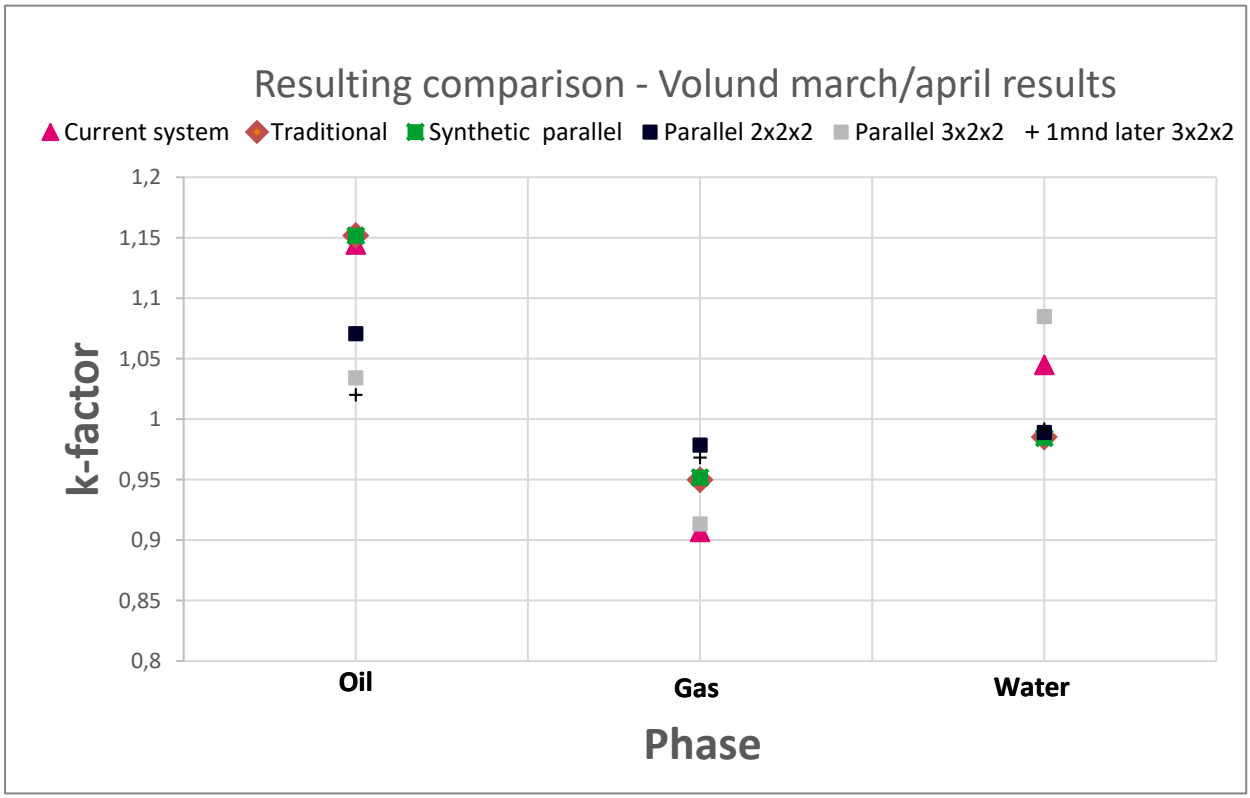


Figure 41 - Resulting comparison of Volund on March April results