Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialization: Bachelor of Science in Energy & Petroleum Engineering	Spring semester, 2023. Open access
Author: Vegard Berg Harlem	_____ (Author's signature)
Supervisor(s): Alf Kristian Gjerstad	
Title of bachelor's thesis: Comparative Analysis and Enhancement of Simplified Drilling Process Simulation Models and Exploring Machine Learning for Real-Time Optimization	
Credits: 20 ECTS	
Keywords: Herschel-Bulkley Power-Law Bingham Plastic Newtonian Shear Stress Yield Point	Number of pages: XI + 53 + supplemental material/other: 89 Stavanger, 12th June 2023

Comparative Analysis and Enhancement of Simplified Drilling Process Simulation Models and Exploring Machine Learning for Real-Time Optimization

By

Vegard Berg Harlem

Bachelor's Thesis

Presented to the Faculty of Science and Technology

The University of Stavanger

THE UNIVERSITY OF STAVANGER

JUNE 2023

Acknowledgement

I would like to extend my deepest gratitude to my supervisor, A. Kristian Gjerstad, for his unwavering support, guidance, invaluable insights, and expertise throughout my academic journey.

Additionally, I would like to convey my heartfelt thanks to my classmates and colleagues for creating a nurturing and positive atmosphere at the workspace. Their support and collective enthusiasm have fostered an environment conducive to collaboration and the nurturing of personal development.

Finally, I am grateful to the University of Stavanger that supported this academic venture, providing the necessary resources for its successful completion.

Abstract

Real-time optimization of drilling processes is vital for the efficient and safe operation of the oil and gas industry. For this, fast and robust models are required to enable automation safety strategies. Many existing models are computationally intensive while requiring executions speeds decades faster than real-time for certain automation tasks. This thesis aims to understand what makes models computationally intensive, compare solutions and propose alternatives, as well as look at accuracy where simplifications are made.

The research framework involves two models developed in MATLAB, by Alf Kristian Gjerstad and Kjell Kåre Fjelde, as a starting point. The primary tasks include analyzing the differences between the models mainly aimed at calculation of frictional pressure loss, evaluating the reasons for these differences, modifying the models to suit the needs of this thesis, and adding options for the calculations in the main model, by Alf Kristian Gjerstad.

This thesis presents a thorough investigation of the discrepancies between the two models, along with implementations and modifications to the main model. A MacineLearning-based approach is proposed as an alternative to the more computationally intensive versions using Herschel-Bulkley and Bingham Plastic, to maintain real-time applicability while hopefully maintaining accuracy. The results demonstrate the potential of the proposed alternative.

Acronyms

ODE – Ordinary Differential Equation

PDE – Partial Differential Equation

RMSE – Root Mean Square Error

List of Contents

List of Figures

List of Tables

1 Introduction

The oil and gas industry relies heavily on drilling processes to access and extract valuable resources from the earth. With the growing demand for energy and the increasing complexity of drilling operations, there is a pressing need for real-time optimization and automation of these processes. Real-time optimization can reduce operational costs, increase efficiency, and enhance safety for both equipment and personnel. To achieve this, fast, robust, and accurate models that can simulate these processes in real-time or faster is essential.

Modern models for drilling process simulation often involve complex and computationally intensive calculations. While these models provide a high level of accuracy, their computational demands often make them unsuitable for real-time application. This has led to a growing interest in the development of computationally efficient models that can still provide accurate representations of the drilling processes.

1.1 Objective

This thesis aims to address the challenges of developing a simplified model for drilling process simulation, with a particular focus on the laminar case in an annulus using slot approximation for estimation. The model strives to balance computational efficiency with accuracy, primarily in the context of friction pressure loss calculations. The research will explore the differences between various models, including Newtonian, Power law, Bingham Plastic, and Herschel-Bulkley, in their approach to these calculations. A machine learning model will be proposed as a potential solution to the computational intensity associated with these calculations. The performance of this machine learning model, in terms of both accuracy and efficiency, will be thoroughly evaluated.

1.2 The provide well simulation models

In the appendices of my thesis, two distinct MATLAB models are presented: the Alf Kristian Gjerstad Model (Appendix A) and the Kjell Kåre Fjelde Model (Appendix B). Both models are designed to perform complex calculations related to well dynamics, but they employ different methods and functions to achieve their results.

1.3 Alf Kristian Gjerstad Model (Appendix A)

The Gjerstad Model is a comprehensive computational framework designed to simulate and analyze various aspects of well dynamics. It primarily utilizes an Ordinary Differential Equation (ODE) approach, providing a balance between computational efficiency and accuracy. The model includes key scripts for setting up and running the simulation, adjusting parameters, and generating results. These scripts handle tasks such as initializing the simulation, setting mud type parameters, solving the ODEs, calculating the rate of change of fluid flow and pressure, and advancing the simulation by one time step. The Gjerstad Model is a robust tool for understanding and predicting well dynamics.

The model also includes a suite of functions for initializing and setting up the simulation (PipeFluGen_2xOrd_Init.m, PipeFluHrz_2xOrd_Setup.m), as well as functions for running the simulation and generating results (Solver_RK4_New.m, PipeFluHrz_Step.m).

1.4 Kjell Kåre Fjelde Model (Appendix B)

The Fjelde Model, in contrast, is a Partial Differential Equation (PDE) model that employs a distinct set of functions to achieve similar objectives. This model includes functions for calculating sound speed, frictional pressure drop, and liquid and gas densities. The use of a PDE approach allows the Fjelde Model to capture spatial variations in the system, providing a more detailed and accurate representation of the dynamics. However, it is generally more complex to solve numerically and may require more computational resources.

1.5 Comparison

While both models aim to simulate and analyze well dynamics, they do so using different methods and functions. The Gjerstad Model seems to be more comprehensive, with a wider range of functions for different aspects of the simulation. The Fjelde Model, on the other hand, appears to be more focused on specific calculations related to flow parameters.

The Gjerstad model, an Ordinary Differential Equation (ODE) model, simplifies the complexities of the system into a set of differential equations. These equations are then solved using the Runge-Kutta method, a powerful numerical technique that provides accurate solutions to the ODEs. This approach allows for a more straightforward numerical solution and interpretation. However, it may not capture all the intricate dynamics of the system, especially when spatial variations are significant.

On the other hand, the Fjelde model is a Partial Differential Equation (PDE) model. It considers spatial variations in the system, providing a more detailed and accurate representation of the dynamics. However, PDE models are generally more complex to solve numerically and may require more computational resources. The choice between these models depends on the specific requirements of the study, such as the level of detail needed and the available computational resources.

In conclusion, both models are complex and sophisticated tools for analyzing well dynamics. They each offer unique methods and functions that can be utilized to understand and predict the behavior of wells.

1.6 Starting point

In order to establish a solid starting point for the development and analysis of the models, it is crucial to identify and define the relevant parameters that influence the calculations of friction pressure loss. Table 1 lists these parameters, where they are found and the chosen initial value for comparison.

Table 1 Parameters for initial comparison

Parameters	Kjell Kåre Fjelde	Line	Alf Kristian Gjerstad	Line	Value	Units	Comment
Sim Time	main17042023.m	59	MasterAlg_PipeHorizontal.m	42	300	[s]	
Length	main17042023.m	34	MasterAlg_PipeHorizontal.m	119	4000	[m]	
Annular width	main17042023.m	161 and 162	Tw_stringNewtonianLaminar	54	0.102	[m]	Added to Tw_stringNewtonianLaminar for slot approximation
Diameter	main17042023.m	161 and 162	MasterAlg_PipeHorizontal.m	123	0.127	[m]	
Temperature	main17042023.m	76 and 77	N/A	N/A	15	[°C]	
Water density	main17042023.m	97 and 109	SetPhysicsParam.m	12	1000	[kg/m^3]	
Pressure STC	main17042023.m	101 and 110	SetPhysicsParam.m	10	0	[Pa]	
Temperature STC	main17042023.m	102 and 111	N/A	N/A	15	[°C]	
Viscosity	main17042023.m	115	MasterAlg_PipeHorizontal.m	74	0.5	[Pa*s]	
Gravity	main17042023.m	124	SetPhysicsParam.m	9	0	[m/s^2]	
Flowrate	main17042023.m	336	PipeFluHrz_InputSignalGenerator.m	10	2000	[LPM]	
Yield point	N/A	N/A	MasterAlg_PipeHorizontal.m	74	1	[Pa]	
Flow behavior index	N/A	N/A	MasterAlg_PipeHorizontal.m	74	0.8	N/A	
Consistency index	N/A	N/A	MasterAlg_PipeHorizontal.m	74	0.2	[Pa*s]	

In the subsequent sections, the simplified model will be further developed and analyzed, with a focus on understanding the impact of each parameter on the friction pressure loss calculation and the overall performance of the model.

1.7 A More detailed look at the Gjerstad Model (Appendix A)

A.1 MasterAlg_PipeHorizontal.m:

This is the main script in the Gjerstad Model, it sets input parameters, executes the simulation, and plots the results. The script includes sections for setting boundary conditions, executing the simulation for different types of flow models (ODE and PDE), and plotting the results. The simulation results include flow, pressure, and density. The script also includes progress tracking and a completion message.

A.5 fRampAndHold3.m:

This script adjusts the flow rate, pressure, and throttle input based on time. It then executes a step function for the PipeFluHrz object and updates the output vector and plot matrix.

A.10 PipeFluGen_2xOrd_Init.m:

initializes a generic pipe object for the fluid inside it. This function sets up simulation constants, grid parameters, and fluid properties. It also prepares initial conditions for flow rate, pressure, and density. The function is designed to be wrapped by an outer function that provides specific parameters like initial conditions, inclination, and form of inputs/outputs. The reference frame for the fluid is the moving solid pipe, considering the solid string acceleration as a fictitious force.

A.14 SelectMudTypeOrSetParameters.m:

Sets the parameters for a given mud type. It takes in a mud name and either selects pre-set parameters for known mud types or sets custom parameters based on the inputs. The parameters include the consistency index, flow behavior index, and yield point.

A.18 Solver_RK4_New.m:

This function is a Runge-Kutta 4 solver for any ordinary differential equation (ODE) model. It's used for numerical integration to solve the ODEs.

A.20 DsMain_Horizontal_2xOrd_ODE.m:

This script defines a function that calculates the rate of change of fluid flow and pressure inside the drill string. It takes into account various factors such as the acceleration of the drill string, the pressure at the downstream end of the string, the flow rate at the upstream end, and the percentage of throttle closure. The function returns the rate of change of fluid flow and pressure as a vector.

A.23 PipeFluHrz_2xOrd_Setup

Sets up the parameters for the fluid flow simulation in a horizontal pipe. It takes as input the initial conditions, the fluid properties, and some global constants. The function then initializes the state variables (flow rate, pressure, and density) and sets up the parameters for the simulation. It also prepares the data for plotting the results of the simulation.

A.24 PipeFluHrz_InputSignalGenerator.m:

This function generates smooth input signals for the simulation, generating flow rate and throttle input signals.

A.26 PipeFluHrz_Step.m:

This function advances the simulation by one time step. It takes the current state of the system and the inputs for the next time step, and uses a solver to compute the new state of the system. It also updates the state variables and computes additional simulation variables.

A.27 Tw_stringNewtonianLaminar.m:

This function is a placeholder or "dummy" function, designed to calculate the wall shear stress for Newtonian fluids in laminar flow in a pipe. Currently, it is set to return the negative of fluid velocity. However, it is intended to be replaced or modified with a more accurate or complex calculation as needed.

The remaining scripts in the appendix are not central to the thesis. They perform tasks that are either not relevant or not utilized in the main discussion. While they contribute to the overall codebase, their specific roles do not directly influence the thesis outcomes. Therefore, they are not discussed in detail but are included in the appendix for reference.

1.8 A More detailed look at the Fjelde Model (Appendix B)

B.1 main17042023

This is the main code for the Fjelde model, the code simulates two-phase (gas and liquid) flow in a wellbore. It starts by defining the physical and numerical parameters, such as fluid properties and wellbore geometry. It then initializes the variables for pressures, densities, and velocities. The code sets up a time-stepping loop, computing slope limiters to prevent spurious oscillations in the solution. The simulation calculates fluxes between cells and updates the conservative variables accordingly. It also computes source terms for effects like friction and hydrostatic pressure. After updating the variables, it computes physical variables from the conservative ones and calculates phase velocities. The code saves time-dependent variables for later analysis, providing a comprehensive understanding of two-phase flow behaviour in a wellbore.

For this case, the gravitational constant is set to zero to model a horizontal well, and the gas flow is also set to zero. This simplifies the model by removing the influence of gravity and focusing solely on the liquid phase. The absence of gas flow reduces complexity, making it easier to study specific aspects of liquid flow dynamics in a horizontal wellbore setting.

B.2 csound.m:

Calculates mixed sound velocity based on gas volume fraction, liquid density, and pressure.

B.3 dpfric.m:

Computes frictional pressure loss gradient for two-phase flow using mixture values.

B.4 minmod.m:

Determines slope using a minmod limiter function for stability in numerical methods.

B.5 pm.m:

Implements a flux limiter function for negative velocities.

B.6 pp.m:

Implements a flux limiter function for positive velocities.

B.7 psim.m:

Applies a flux limiter function for negative velocities with a tunable parameter.

B.8 psip.m:

Applies a flux limiter function for positive velocities with a tunable parameter.

B.9 rholiq.m:

Estimates liquid density based on pressure and temperature.

B.10 rogas.m:

Calculates gas density based on pressure and temperature.

2 Methodology

This thesis focuses on the calculation of friction pressure loss in the laminar flow regime within an annulus in a horizontal well. The annulus, a space between two concentric cylinders, is a common scenario in drilling operations. However, due to its complex geometry, direct calculations can be challenging. To simplify this, a narrow slot approximation is used to estimate the annular space. This approximation transforms the annular space into a narrow slot with equivalent hydraulic properties, enabling the application of equations originally derived for simpler geometries. The following sections detail the methodologies employed in the Gjerstad and Fjelde models to calculate friction pressure loss under these conditions.

2.1 The calculation of friction pressure

The models created by Alf Kristian Gjerstad and Kjell Kåre Fjelde are multifaceted and serve various purposes. However, for the scope of this thesis, the focus is on their application in calculating friction pressure loss. In Gjerstad's model, the function A.27 Tw_stringNewtonianLaminar.m is used, while in Fjelde's model, the function B.3 dpfric.m is employed. Both these functions utilize modified versions of the Hagen-Poiseuille equation.

$$\Delta p = \frac{8\mu LQ}{\pi R^4}$$

Equation 1 Hagen-Poiseuille Equation

Where:

- μ is the viscosity of the fluid,
- L is the length of the pipe,
- Q is the flowrate in the pipe,
- R is the radius of the pipe and
- Δp is the pressure difference along the length

In Gjerstad's model, the Hagen-Poiseuille equation is adapted to compute the average shear stress over the walls of the conduit, denoted as τ_w. On the other hand, Fjelde's model uses the equation to calculate the frictional pressure loss gradient, symbolized as Δp_f

Fjelde's model, specifically in the laminar case in B.3 dpfric.m, is based on the Darcy-Weisbach equation. This equation calculates pressure loss in a pipe, considering factors such as the friction factor, the length and diameter of the pipe, the fluid's density, and its velocity.

$$\Delta p = f \frac{L}{D} \frac{\rho v^2}{2}$$

Equation 2

Where:

- ρ is the density of the fluid,
- v is the velocity of the fluid and
- D is the diameter of the pipe

The gradient formula in this model is derived from the Darcy-Weisbach equation and the relationship for laminar flow in annulus considered to be a narrow slot,

$$f = \frac{24}{Re}$$

Equation 3

And raynolds number,

$$Re = \frac{\sigma |v| 2 (R_{outer} - R_{inner})}{\mu},$$

Equation 4

Where:

- R_{outer} is the outer diameter of the annulus and
- R_{inner} is the inner diameter of the annulus

which originates from the Hagen-Poiseuille equation. This relationship involves the Reynolds number, a dimensionless quantity predicting flow patterns in different fluid flow situations. Giving us the gradient formula.

$$\Delta p_f = \frac{2 f \rho v |v|}{R_{outer} - R_{inner}}$$

Equation 5

The process of deriving an equation for τ_w involves manipulating known equations and relationships from fluid dynamics, such as the Hagen-Poiseuille equation and the Darcy-

Weisbach equation. These equations contain variables that are directly related to wall shear stress, such as fluid velocity, pipe diameter, and fluid viscosity.

By rearranging these equations and isolating τ_w, you can express wall shear stress in terms of other known quantities. This algebraic manipulation allows you to calculate τ_w for a given set of conditions, which can be invaluable in predicting and understanding the behavior of fluid flow in a conduit or pipe. Combining equations 3, 4 and 5, we get:

$$\Delta p_f = \frac{12\mu v}{(R_{outer} - R_{inner})^2}$$

$$\Delta p_f = \frac{12\mu v * (R_{outer} + R_{inner})}{(R_{outer} - R_{inner})^2 * (R_{outer} + R_{inner})}$$

$$\Delta p_f = \frac{12\mu v * (R_{outer} + R_{inner}) * \pi}{(R_{outer} - R_{inner})^2 * (R_{outer} + R_{inner}) * \pi}$$

$$\Delta p_f = \frac{6\mu v * (R_{outer} + R_{inner}) * 2\pi}{(R_{outer} - R_{inner}) * \left(R_{outer}^{\ 2} - R_{inner}^{\ 2}\right) * \pi}$$

Equation 6

When considering that A_u, the surface area in contact with liquid, divided by the length is

$$A_u = 2\pi(R_{outer} + R_{inner}),$$

Equation 7

And the cross-sectional area of the fluid is

$$A_f = \pi\,(R_{outer}^{\ 2} - R_{inner}^{\ 2}),$$

Equation 8

This results in:

$$\Delta p_f = \frac{A_u}{A_f}\frac{6\mu}{R_{outer} - R_{inner}}v$$

Equation 9

Then use the relationship for shear stress and the pressure loss gradient,

$$\Delta p_f = \frac{A_u}{A_f}\tau_w, \qquad \text{(Gjerstad, 2014)}$$

Equation 10

to show that this gives us the Newtonian narrow slot approximation for shear stress used in **Error! Reference source not found.**

$$\tau_w = -\frac{6\mu}{R_{outer}-R_{inner}}v \qquad \text{(Gjerstad, 2014)}$$

Equation 11

The negative sign in the equation for τ_w signifies this opposition to the fluid's velocity. It is a convention used to indicate the direction of the shear stress relative to the direction of flow. When calculating or analyzing fluid dynamics, it's essential to consider this negative sign to accurately represent the physical reality of the situation.

To enable comparison, the Newtonian narrow slot approximation equation has been added to the model under A.27 Tw_stringNewtonianLaminar.m. This equation is a simplification often used for laminar flow in narrow annular spaces or slots, where the flow can be approximated as flow between parallel plates. This approximation provides a baseline for comparing the behavior of Newtonian fluids under similar conditions.

The inclusion of this equation allows for an initial comparison between the Newtonian model and the newly incorporated non-Newtonian models (Power Law, Bingham Plastic, and Herschel-Bulkley). By comparing these models, we can better understand the differences in fluid behavior and the implications of these differences for practical applications. This comparison will also serve as a validation check, ensuring that the non-Newtonian models reduce to the Newtonian case under the appropriate conditions.

2.2 Expanding the calculation of shear stress to include Non-Newtonian Fluids

Shear stress is the force per unit area exerted by a fluid flowing over a surface, related to the rate of fluid velocity change. The current model for calculating the shear stress in an annular space is based on the principles of fluid dynamics and is influenced by the Hagen-Poiseuille equation, which describes the pressure loss due to viscous friction in a long, straight pipe. This model assumes a Newtonian fluid behavior, where the shear stress is directly proportional to the shear rate. The constant of proportionality is the viscosity of the fluid, a linear relationship first proposed by Sir Isaac Newton. While this model provides accurate predictions for

Newtonian fluids like water and air, it falls short when applied to non-Newtonian fluids, which exhibit a change in viscosity with the rate of shear strain.

To enhance the versatility and applicability of the model, particularly in industries like drilling or chemical processing where non-Newtonian fluids are commonly encountered, it is necessary to incorporate additional rheological models into the calculations. These include the Power Law, Bingham Plastic, and Herschel-Bulkley models.

The Power Law model, also known as the Ostwald–de Waele relationship, is used for fluids that exhibit shear-thinning or shear-thickening behavior. This behavior is characterized by a decrease or increase in viscosity with the rate of shear strain. The Power Law model captures this behavior with just two parameters: the consistency index (K) and the flow behavior index (n).

The Bingham Plastic model is used for fluids that behave like a solid under low shear stress but flow like a fluid under high shear stress. These fluids have a yield stress that must be exceeded before they start to flow. This model, developed by Eugene C. Bingham, is particularly useful for describing materials like toothpaste and mayonnaise.

The Herschel-Bulkley model is a generalized version of the Bingham Plastic model that also captures shear-thinning or shear-thickening behavior. It adds an additional parameter (n) to the Bingham model, allowing it to describe a wider range of non-Newtonian fluids. This model is often used in the drilling industry to describe drilling muds.

By incorporating these additional models into the shear stress calculations, the model's predictive capabilities will be significantly enhanced for a wider range of fluids. This expansion not only increases the model's accuracy but also its relevance in real-world applications where non-Newtonian fluids are frequently encountered. The following sections will delve into the mathematical formulation of these models and their implementation into the existing framework.

Table 2 Rheology model equations

Rheology Model	**Poiseuille Flow (Stationary Walls)**
Newtonian	$\tau_w = -\frac{6\mu}{R_{outer} - R_{inner}} v$ *Equation 10*
Power-law	$\tau_w = \mp k(\frac{4n+2}{nh}\lvert v\rvert)^n$ *Equation 12*
Bingham plastic	$\lvert\tau_w\rvert > \tau_y$, $v = \frac{-h}{6\mu_p}\tau_w\left[1 + \frac{3}{2}\frac{\tau_y}{\lvert\tau_w\rvert} + \frac{1}{2}(\frac{\tau_y}{\lvert\tau_w\rvert})^3\right]$ *Equation 13*
	$\lvert\tau_w\rvert \leq \tau_y$, $v = 0$
Herschel Bulkley	$\lvert\tau_w\rvert > \tau_y$, $v = \frac{-h}{2k^n\tau_w}\frac{\left(\lvert\tau_w\rvert - \tau_y\right)^{n+1}}{n+1}[1 - \frac{\left(\lvert\tau_w\rvert - \tau_y\right)}{(n+2)\lvert\tau_w\rvert}$ *Equation 14*
	$\lvert\tau_w\rvert \leq \tau_y$, $v = 0$

(Gjerstad, 2014)

Table 2 presents the equations for different rheology models under Poiseuille flow with stationary walls. These models describe the behavior of different types of drilling fluids, and the equations represent the relationship between wall shear stress (τ_w) and fluid velocity (v). The parameters in these equations are as follows:

- μ: viscosity in the Newtonian model.
- R_{outer} and R_{inner}: Outer and inner radii of the annulus.
- k and n: Consistency index and flow behavior index in the Power-law model.
- τ_y: Yield stress in the Bingham plastic and Herschel-Bulkley models.
- μ_p: Plastic viscosity in the Bingham plastic model.

- h: The distance between the walls of the annulus.
- τ_w: Shear stress

A defining characteristic of Bingham Plastic and Herschel-Bulkley fluids is the presence of a yield stress, denoted as τ_y. This yield stress is the minimum shear stress that must be applied to the fluid before it begins to flow. Below this threshold, the fluid behaves like a solid and does not flow, hence the fluid velocity is zero.

The Bingham plastic and Herschel-Bulkley models, represented by equations 13 and 14, cannot be directly rewritten for wall shear stress (τ_w). This complexity makes it challenging to express these models directly in terms of τ_w, requiring advanced numerical methods or approximations for solutions.

The concept of yield stress arises from the internal structure of these fluids. They consist of particles or structures that form a network, providing the fluid with a certain degree of internal resistance to flow. When the applied shear stress is less than the yield stress ($\tau_w \leq \tau_y$), this network remains intact, and the fluid does not flow. It behaves more like a solid, maintaining its shape unless subjected to stress greater than the yield stress.

In the context of the Bingham Plastic model, once the yield stress is exceeded, the fluid behaves like a Newtonian fluid with a constant viscosity. For the Herschel-Bulkley model, the fluid exhibits shear-thinning or shear-thickening behavior once the yield stress is exceeded, depending on the value of the flow behavior index.

This behavior has significant implications for the flow of these fluids in pipes or annular spaces. If the shear stress applied by the pressure gradient is less than the yield stress, there will be no flow. This can lead to a plug of stationary fluid, which can be a challenge in applications like drilling or pumping of suspensions. Understanding and accurately modeling this behavior is crucial for the design and operation of processes involving Bingham Plastic or Herschel-Bulkley fluids.

2.3 Machine learning

Machine learning is a subset of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. It focuses on the development of computer programs that can access data and use it to learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers to learn automatically without human intervention or assistance and adjust actions accordingly. Machine learning algorithms are often categorized as supervised or unsupervised.

The focus of this thesis will be on supervised machine learning, where algorithms can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient training.
(Wikipedia, 2023)

2.4 Random Forest Regression

Random Forest is a versatile machine learning method capable of performing both regression and classification tasks. It also undertakes dimensional reduction methods, treats missing values, outlier values, and other essential steps of the data exploration, and does a fairly good job. It is a type of ensemble learning method, where a group of weak models combine to form a powerful model.

In Random Forest Regression, a dependent variable is predicted using multiple decision trees. Each decision tree is constructed by using a subset of the data and variables, and the average prediction of all the trees is considered as the final prediction. This method helps to overcome the problem of overfitting, which is a modelling error that occurs when a function is too closely fit to a limited set of data points, making it capture the noise in the data, which is common in decision tree models.

Random Forest Regression works in four basic steps:

1. Selection of random samples from a given dataset.
2. Construction of a decision tree for each sample and getting a prediction result from each decision tree.
3. Voting for each predicted result.
4. Select the prediction result with the most votes as the final prediction.

(Wikipedia, 2023)

2.4.1 Decision Trees

A decision tree is a fundamental component of a random forest and is used as a predictive model in machine learning. It maps observations about an item to conclusions about the item's target value. Essentially, decision trees are used for making decisions and predictions by mapping out a set of rules that lead to a certain outcome based on input data.

The decision tree model follows a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model. The structure of a decision tree includes nodes and branches. The topmost node, known as the root, represents the entire population or sample, and this gets divided into two or more homogeneous sets. The end nodes of the tree, known as leaves, represent the decisions or predictions.

One of the main advantages of decision trees is their simplicity and interpretability - they can be easily visualized and understood. However, they can suffer from overfitting, where the model captures the noise in the data and becomes too complex, leading to poor predictive performance on unseen data. This is where Random Forest Regression comes in, using multiple decision trees and averaging their predictions to achieve a more robust and accurate model.

2.4.2 Root Mean Square Error

The Root Mean Square Error (RMSE) is a frequently used measure of the differences between values predicted by a model and the values actually observed. It is a standard way to measure the error of a model in predicting quantitative data. The RMSE represents the square root of the second sample moment of the differences between predicted values and observed values or the quadratic mean of these differences. These deviations, called residuals when the calculations

are performed over the data sample that was used for estimation, are also called prediction errors when computed out-of-sample. The RMSE serves to aggregate the magnitudes of the errors in predictions into a single measure of predictive power. A lower RMSE is indicative of a better fit to the data.

The formula for RMSE is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_o - y_p)\text{^}2}$$

Equation 15

Where:

- n is the number of observations,
- y_o is the observed value,
- y_p is the predicted value.

(Wikipedia, 2023)

3 Results and Discussion

3.1 The implementation of the different shear stress calculations

In this thesis, new code has been developed to implement various shear stress calculations. These calculations are integral to the simulation of fluid flow and pressure loss in the drilling process. The code incorporates different rheological models and applies the narrow slot approximation to estimate the annular space in a horizontal well. The implemented code is provided in Appendix C.

3.1.1 Tw_NarrowSlotNewtonianLaminar.m

In the Newtonian case, the shear stress calculations were performed using a narrow slot approximation for wall shear stress (τ_w) under laminar flow conditions. The calculation was implemented using the MATLAB code in C.1 Tw_NarrowSlotNewtonianLaminar.m.

In this function, the wall shear stress (τ_w) is calculated based on the fluid viscosity (mu), the cross-sectional area of the geometry (AreaCrs calculated as shown in D.1 PipeFluGen_2xOrd_Init.m), the flow rate, the pipe velocity (VelocityPipe which is set to 0 for all these functions for simplification), and the slot height (h). The fluid velocity is calculated by dividing the flowrate by the cross-sectional area.

$$V = \frac{Q}{A_f}$$

Equation 16

The wall shear stress is then calculated as -6 times the fluid viscosity divided by the slot height, all multiplied by the fluid velocity as shown in equation 10.

The implementation of this function was straightforward due to the simplicity of the formula, which resulted in efficient calculations. The function was able to quickly calculate the wall shear stress for the Newtonian case under laminar flow conditions, demonstrating the efficiency of traditional methods for this case.

3.1.2 Tw_NarrowSlotPowerLawLaminar.m

For the Power Law case, the shear stress calculations were also performed using a narrow slot approximation for wall shear stress (τ_w) under laminar flow conditions. However, the Power Law case involves a non-Newtonian fluid, which follows a different viscosity model. The calculation was implemented using the MATLAB function from C.2 Tw_NarrowSlotPowerLawLaminar.m

In this function, the wall shear stress (τ_w) is calculated based on the fluid viscosity (mu), the cross-sectional area of the geometry (AreaCrs), the flow rate, the slot height (h), the consistency index (k), and the flow behavior index (n). The fluid velocity is calculated using equation 12

Similar to the Newtonian case, the implementation of this function was straightforward due to the simplicity of the formula, which resulted in efficient calculations. The function was able to quickly calculate the wall shear stress for the Power Law case under laminar flow conditions, demonstrating the efficiency of traditional methods for this case as well.

3.1.3 Tw_NarrowSlotBinghamPlasticLaminar.m

For the Bingham Plastic case, the shear stress calculations were more complex. This case involves a non-Newtonian fluid that exhibits a yield stress (τ_y), meaning it behaves like a solid until this yield stress is exceeded. A simplified version of the Bingham Plastic model was used for the initial guess, which could potentially introduce inaccuracies. Specifically, the wall shear stress (τ_w) might not always be higher than the yield stress, even though the simplified Bingham Plastic model assumes it to be, and vice versa. Furthermore, the simplified Bingham Plastic model was used when τ_w was less than or equal to τ_y, which could introduce noise into the calculations that does not completely correct itself until steady state is achieved.

The calculation was implemented using C.3 Tw_NarrowSlotBinghamPlasticLaminar.m. In this function, the wall shear stress (τ_w) is calculated based on the yield stress (τ_y), the fluid viscosity (mu), the length per grid (L), the slot height (h), the cross-sectional area of the geometry (AreaCrs), and the flow rate. The fluid velocity is calculated by subtracting the pipe velocity from the flow rate divided by the cross-sectional area.

The initial guess for the wall shear stress is calculated using the simplified Bingham Plastic model, as shown in Equation 17.

$$\tau_w = \tau_y + \frac{\mu_p 6Q}{h^3}$$

Equation 17

Which is a simplification based on the bingham plastic model,

$$\tau_w = \tau_y + \mu_p \dot{\gamma}$$

Equation 18

The shear rate $\dot{\gamma}$, is defined as the rate of change of velocity with respect to the distance perpendicular to the flow direction. For a narrow slot or channel, this can be approximated as the average velocity divided by the half-height of the slot or channel. So,

$$\dot{\gamma} = \frac{2v_{avg}}{h} = \frac{2Q}{wh^2}$$

Equation 19

If we further assume that the width of the slot or channel is much larger than the height (which is often the case for a narrow slot or channel), then we can ignore w in the denominator, and the shear rate becomes approximately:

$$\dot{\gamma} = \frac{6Q}{h^3}$$

Equation 20

In this case, the initial guess should never be less than the yield stress (τ_y), unless the flow becomes negative. However, as this is one of the conditions for the calculation shown in Table 2, the condition is kept in the function. If the absolute value of this initial guess is greater than the yield stress, the function solves an equation for τ_w using the fsolve function.

The methodology used for the Bingham Plastic case, specifically the decision to set the wall shear stress (Tw) to the value calculated by the simplified Bingham Plastic model when Tw is less than or equal to the yield stress (τ_y), may not be the most accurate representation of the behaviour of a Bingham Plastic fluid. This approach assumes that the fluid behaves according to the simplified Bingham Plastic model under these conditions, which may not fully capture the behaviour of a Bingham Plastic fluid. As such, this methodology could introduce some inaccuracies into the calculations and results in a noisy output that does not completely correct itself until steady state is achieved.

While the current methodology provides a starting point for calculations, it doesn't fully capture the behavior of a Bingham Plastic fluid, which doesn't flow until the yield stress is exceeded. However, other areas of the model address this by setting the flow rate to zero when the wall shear stress is less than or equal to the yield stress.

3.1.4 Tw_NarrowSlotHerschelBulkley.m

The Herschel-Bulkley case, implemented using the C.4 Tw_NarrowSlotHerschelBulkleyLaminar function, involves a more complex model due to the unique behavior of Herschel-Bulkley fluids. These fluids exhibit a yield stress (τ_y), behaving like a solid until this yield stress is exceeded. The initial guess for the wall shear stress (τ_w) is calculated using a simplified Bingham Plastic model, which assumes that τ_w is always greater than τ_y. However, this may not always be the case, potentially introducing inaccuracies into the calculations.

The methodology used in this case sets τ_w to the value calculated by the simplified Bingham Plastic model when τ_w is less than or equal to τ_y. While this approach simplifies the calculations, it may not accurately represent the behavior of a Herschel-Bulkley fluid. This could result in a noisy output that does not completely correct itself until steady state is achieved.

3.2 Efficiency and speed of simulations

The efficiency and speed of the simulations were key considerations in this study. For the Newtonian and Power Law cases, the traditional methods of calculating shear stress proved to be highly efficient due to the simplicity of the formulas used. The MATLAB functions implemented for these cases were able to quickly perform the calculations, resulting in fast simulation times. Specifically, the Newtonian case averaged about 1.4 seconds per simulation run, while the Power Law case averaged slightly longer at about 1.6 seconds per run. These quick runtimes demonstrate the computational efficiency of these traditional methods in handling the calculations for these cases.

In order to evaluate the efficiency and accuracy of the developed models, it is beneficial to compare them with a well-established model in the field. The Fjelde model serves as an appropriate baseline for this comparison.

When comparing the developed models with the Fjelde model, several key parameters should be considered:

- Computational Speed: The time it takes for a simulation to run is a critical factor in real-time drilling operations. The faster the simulation, the more useful it is in making timely decisions. Therefore, the average runtime of the simulations for each model should be compared. The runtime is measured using the tic toc operator in MATLAB. Running on a 3.8 GHz machine
- Accuracy: The accuracy of the models in predicting the friction pressure loss is crucial. The models should be compared in terms of how closely their predictions match the calculated steady states. The ramp will be evaluated visually.
- Robustness: The models should be able to handle a variety of drilling conditions and fluid properties. They should be tested under different scenarios to see how well they perform and how stable their predictions are.

By comparing the models in terms of these parameters, we can gain a better understanding of their strengths and weaknesses and identify areas for further improvement.

3.2.1 The Fjelde Model

Before comparing the developed models, it is essential to establish a baseline for comparison. This baseline is provided by the Fjelde model, a well-established model in the field of drilling process simulation. The Fjelde model was evaluated in terms of computational speed, accuracy, and robustness, using the initial values listed in Table 1.

3.2.1.1 Computational Speed

The Fjelde model's computational speed was assessed by recording the average runtime for simulations using the initial values from Table 1. This provides a benchmark against which the runtimes of the developed models can be compared. When running the Fjelde model, the average runtime sits at around 2 seconds, varying from about 1.95 seconds to about 2.1 seconds.

3.2.1.2 Accuracy

The accuracy of the Fjelde model was evaluated by comparing its predictions of friction pressure loss with actual measurements, using the initial values from Table 1. This comparison provides a measure of the model's predictive accuracy, which is crucial for effective drilling operations. Figure 1 is a plot of the friction pressure and flowrate.

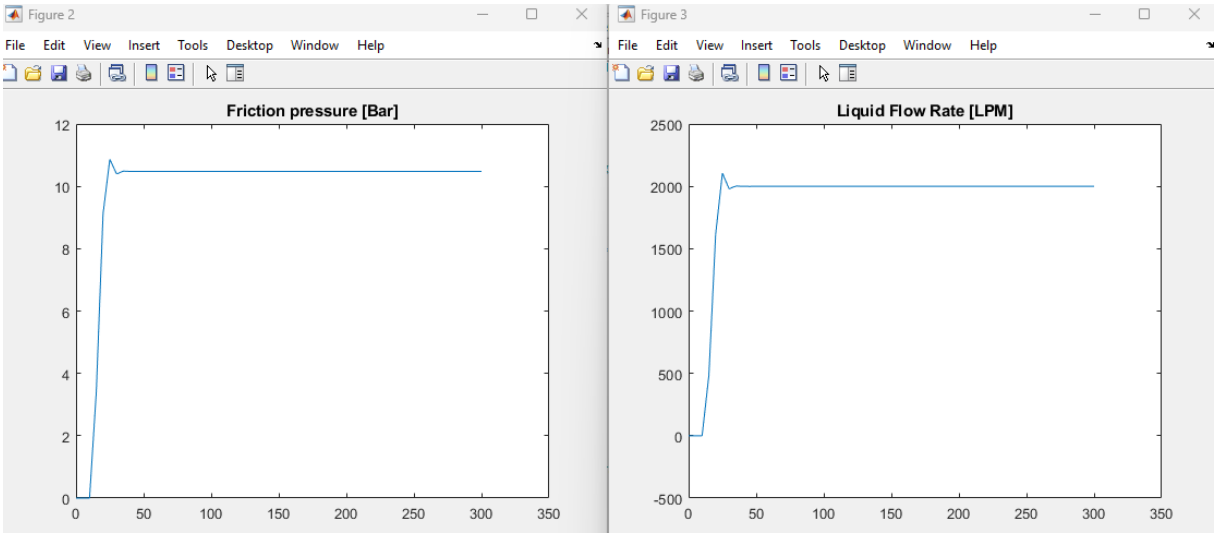

Figure 1 The Fjelde Model

The friction pressure following a similar curve to the flowrate aligns with expected fluid dynamics behaviour. The steady state value of the flowrate aligns with the Hagen-Poiseuille equation, reinforcing the accuracy of the model in simulating laminar flow conditions. To show this, the equations in 2.1, which results in a pressure difference of about 10.48 bar, from a frictional loss gradient of about 262 [pa/m], which also coincides with what the model is providing through B.3 dpfric.m.

3.2.1.3 Some values to display robustness

The robustness of the Fjelde model was assessed by testing it under a variety of drilling conditions and fluid properties, using the initial values from Table 1. The model's performance under these different scenarios provides an indication of its stability and reliability. The condition for the tests of robustness are shown in Table 3

Table 3 Parameters for Robustness

Figure Fjelde Model	2	3	4	5	6
Figure Gjerstad Model	8	9	10	11	12
Sim Time	300	100	100	100	1000
Flowrate [LPM]	2000	10000	2000	2000	2000
Length [m]	1000	4000	4000	4000	10000
Diameter inner [m]	0.127	1	0.127	0.127	0.127
Diameter outer [m]	0.331	2	0.331	0.13	0.331

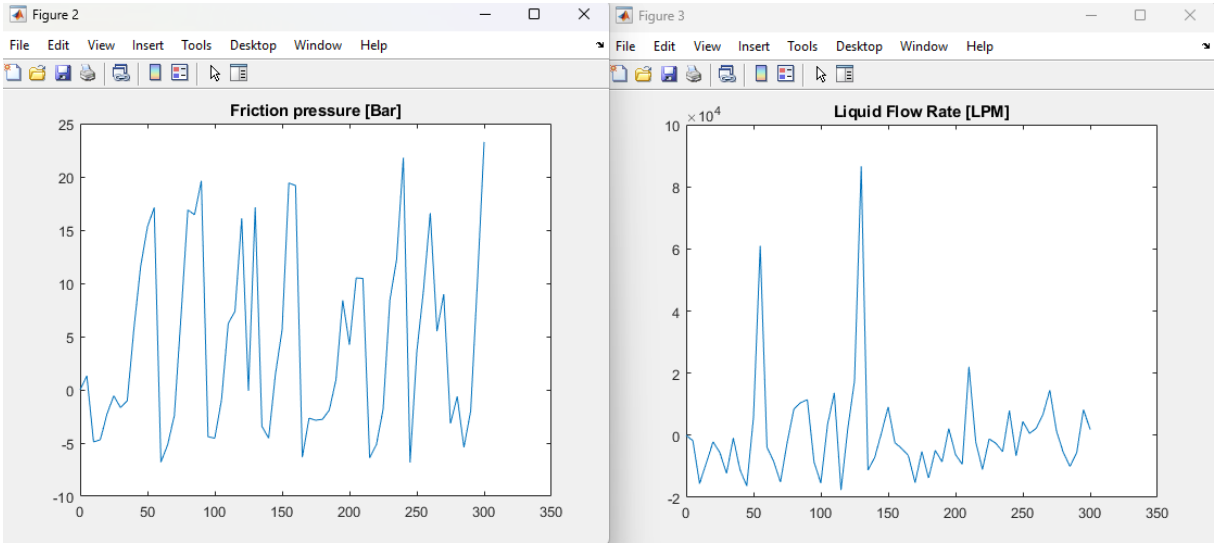

Figure 2

Figure 2 reveals that the model struggles with shorter wells, exhibiting significant inaccuracies and inconsistencies in the simulation results for these cases.

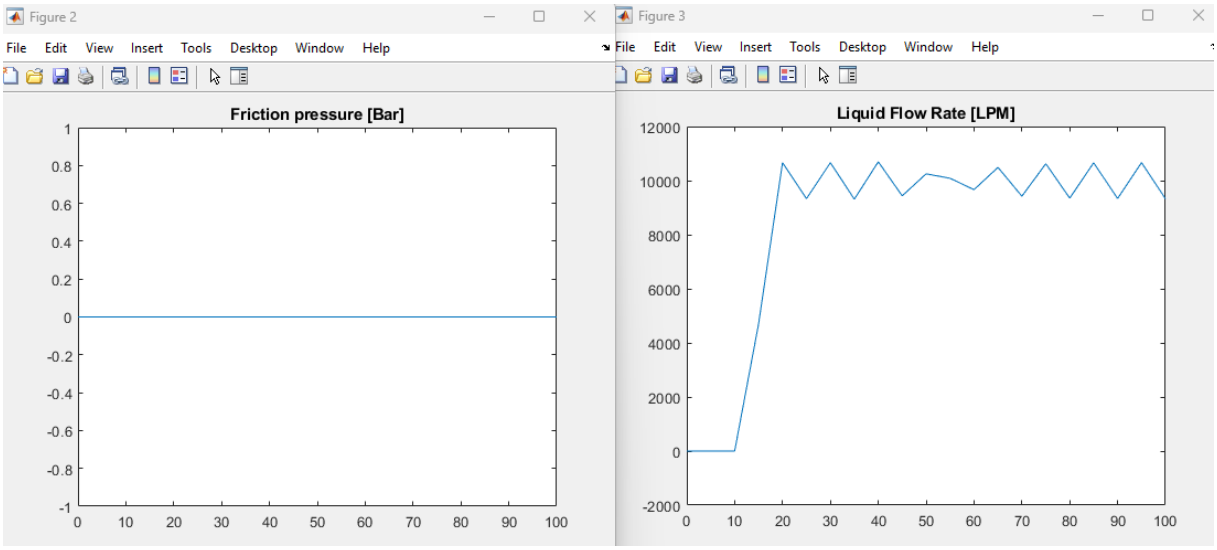

Figure 3

Figure 3 reveals that a large annular gap will reduce the frictional pressure gradient to 0, this is caused by Reynolds number, calculated in B.3 dpfric.m, never getting high enough, and is intended. For this case one could consider the liquid to never fill the annulus, rather flowing like an open river.

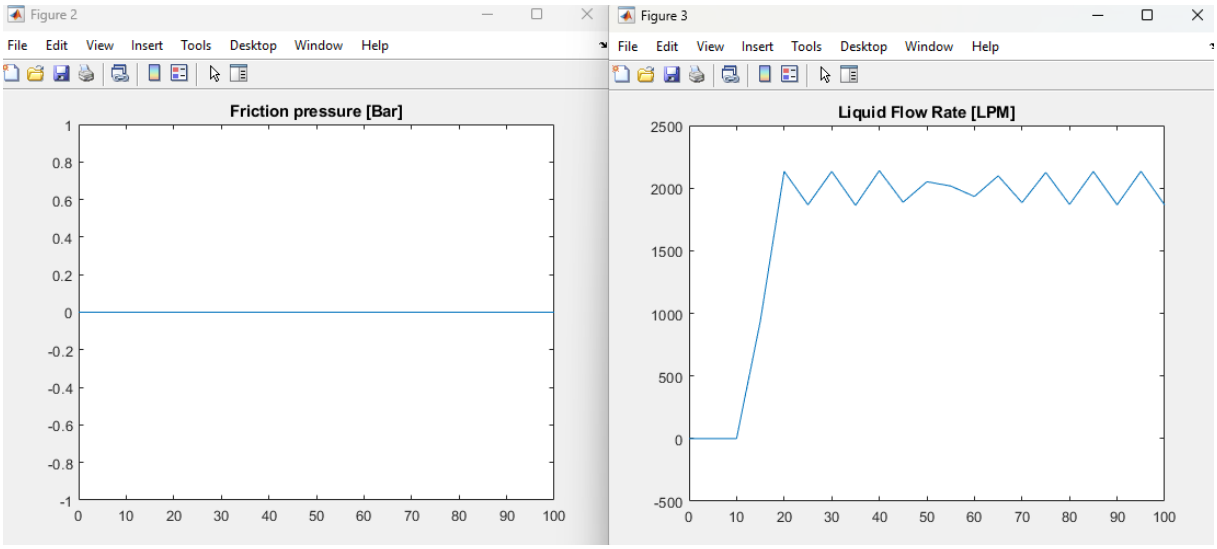

Figure 4

Figure 4 reveals that when the viscosity of the fluid is sufficiently low, the behavior of the fluid in the annulus changes significantly. The friction in the annulus decreases due to the reduced resistance to flow. This is because viscosity is a measure of a fluid's resistance to shear or flow, and a lower viscosity means the fluid can flow more easily. This is also a scenario where Reynolds number is too low for a frictional pressure to build.

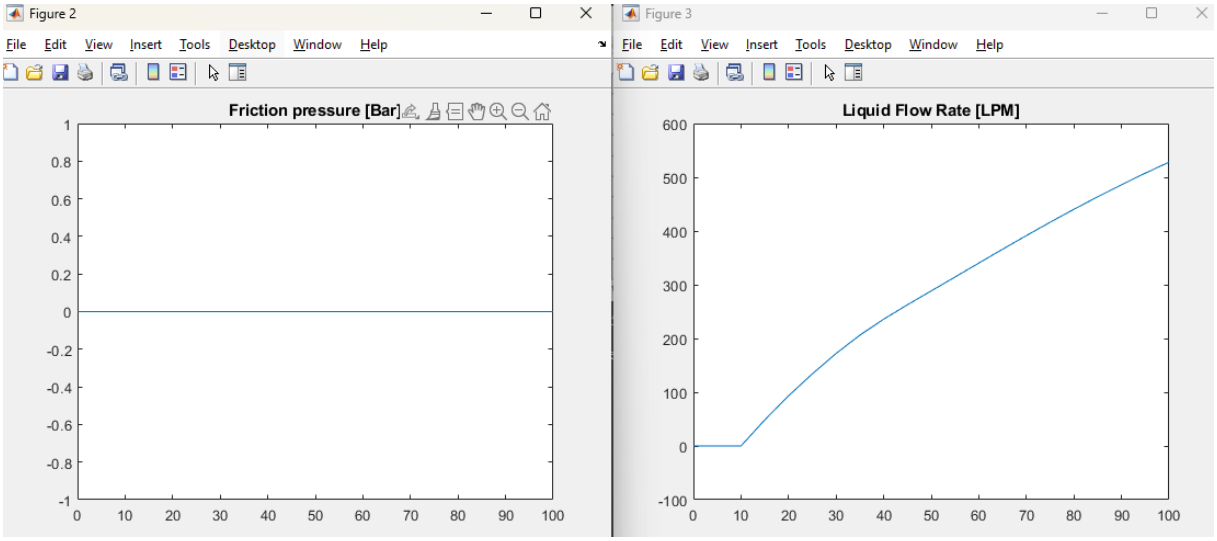

Figure 5

Figure 5 presents an intriguing observation. As anticipated, a smaller annular gap restricts fluid flow due to the reduced space. However, the results deviate from expectations, suggesting a potential limitation of the model. In reality, even with a very small annular gap, the frictional pressure would not be zero due to the fluid's viscosity and the surface roughness of the annulus. Furthermore, such a small annular gap is not typically encountered in practical drilling operations, adding another layer of complexity to the interpretation of these results. The model's inability to accurately simulate these conditions could be interpreted as a shortfall.

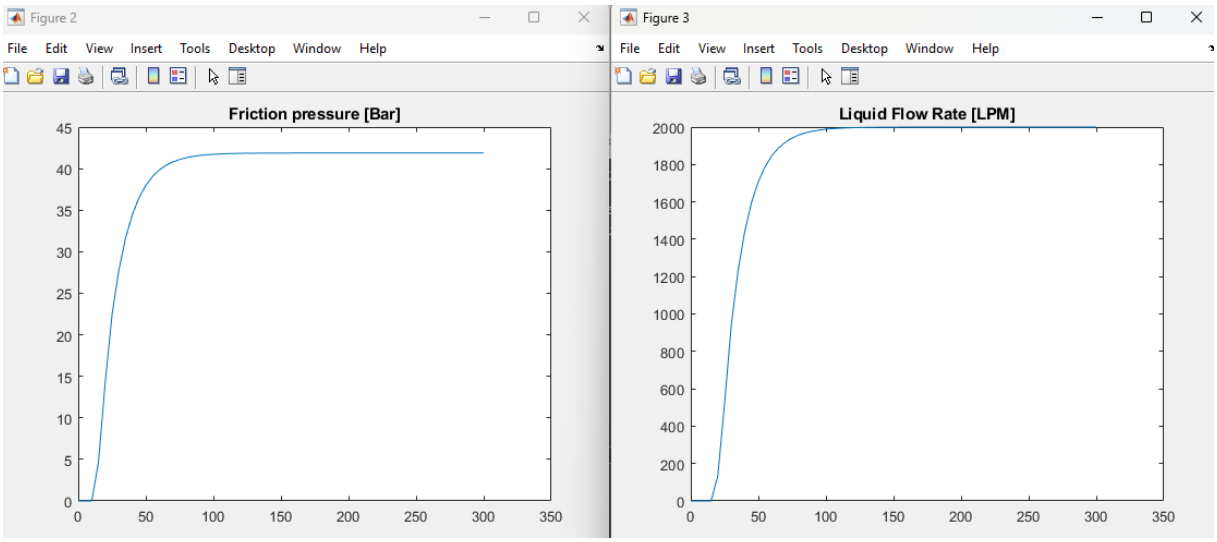

Figure 6

Figure 6 demonstrates a simple case, and as expected the model handles it as it should.

3.2.2 The Gjerstad Model

In the development of the Gjerstad model, we will begin with the Newtonian case, utilizing the function C.1 Tw_NarrowSlotNewtonianLaminar. This function calculates the wall shear stress (τ_w) for a Newtonian fluid in a narrow slot under laminar flow conditions. Starting with the Newtonian case is a strategic choice, as it is expected to yield results that align with the Fjelde model. This is due to the fact that both models are based on the same fundamental principles of fluid dynamics and share the same assumptions for Newtonian fluids. Therefore, any discrepancies between the results of the two models can be attributed to differences in their implementation or the specific conditions under which they are applied, rather than differences in the underlying physics. This approach will provide a solid foundation for the development of the Gjerstad model and facilitate a meaningful comparison with the Fjelde model.

In order to maintain consistency and facilitate a direct comparison with the Fjelde model, the initial conditions for the Gjerstad model are taken from Table 1. Which were used in the Fjelde model simulations. By using the same initial conditions, we can ensure that any differences observed in the results of the two models are due to the models themselves and not variations in the input parameters. This approach allows for a fair and accurate evaluation of the performance and accuracy of the Gjerstad model in comparison to the Fjelde model.

3.2.2.1 Computational Speed

The computational speed of the Gjerstad model was also evaluated using the same initial values from Table 1. This allowed for a direct comparison with the Fjelde model. The average runtime for the Gjerstad model was found to be approximately 1.87 seconds, with a range from about 1.82 seconds to 1.95 seconds, when run to display 4 sections, as shown in Figure 7, and approximately 1.35 seconds with a range from about 1.32 seconds to 1.4 seconds, when run to display one section.

The Gjerstad model, with its comprehensive framework as a representation of the drilling process, has a shorter average runtime than the Fjelde model. This is largely due to the physics simplifications inherent in its design as an Ordinary Differential Equation (ODE) model. These simplifications reduce the complexity of the calculations being executed, thereby enhancing computational efficiency.

On the other hand, the Fjelde model, being a Partial Differential Equation (PDE) model, offers a more detailed representation but at the cost of increased computational time. The choice between models ultimately depends on the balance between the need for computational efficiency and the level of detail required in the simulation.

3.2.2.2 Accuracy

The Gjerstad model, while employing simplified physics in its design as an Ordinary Differential Equation (ODE) model, still offers accuracy comparable to the Fjelde model. The simplifications in the Gjerstad model allow for efficient computations while maintaining a level of accuracy that is suitable for practical applications in drilling process simulations.

The calculations outlined in Section 2.1 yield results that are consistent with those of both the Gjerstad model, and the Fjelde model, further supporting the accuracy of the Gjerstad model. This is evident when comparing the green line in the Gjerstad model's output, as shown in Figure 7, which represents the last section of the annular space, with the graphed pressure in the Fjelde model, as shown in Figure 1. The close alignment of these two lines indicates that the Gjerstad model is able to accurately replicate the results of the Fjelde model, while also providing additional detail and insights.

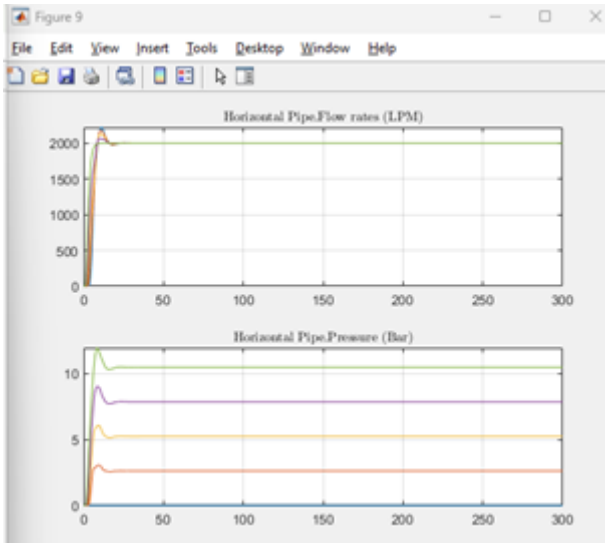

Figure 7

3.2.1.3 Some values to display robustness

To further enhance the comparison between the Gjerstad and Fjelde models, the values from Table 3, which were used in the Fjelde model, will also be applied to the Gjerstad model. This

will provide a more direct comparison of the two models under the same conditions, allowing for a clearer assessment of their relative performance and robustness.

The robustness of a model is a measure of its ability to produce reliable and accurate results under a variety of conditions. By applying the same parameters to both models, we can evaluate how well each model handles changes in the input values and whether they can maintain their accuracy and reliability under these different conditions.

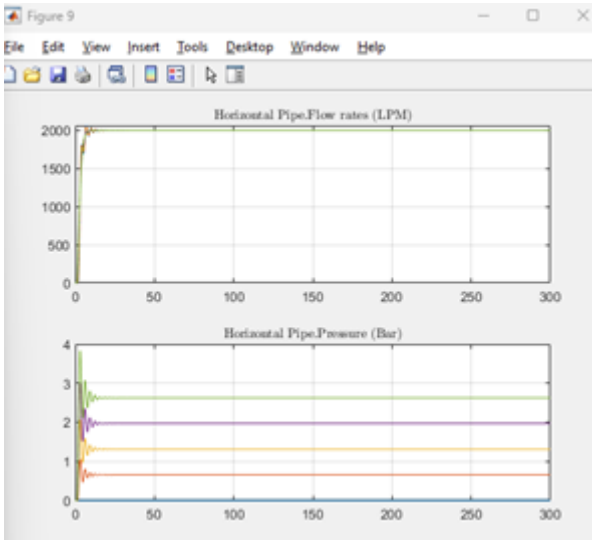

Figure 8

In comparing the performance of the Gjerstad model in Figure 8 and the Fjelde model in Figure 2, it appears that the Gjerstad model handles the conditions more effectively. This could be attributed to the specific parameters or conditions of the case in Figure 8, which may be more suited to the assumptions and calculations inherent in the Gjerstad model. This observation underscores the importance of understanding the strengths and limitations of each model and selecting the most appropriate model based on the specific requirements of the simulation.

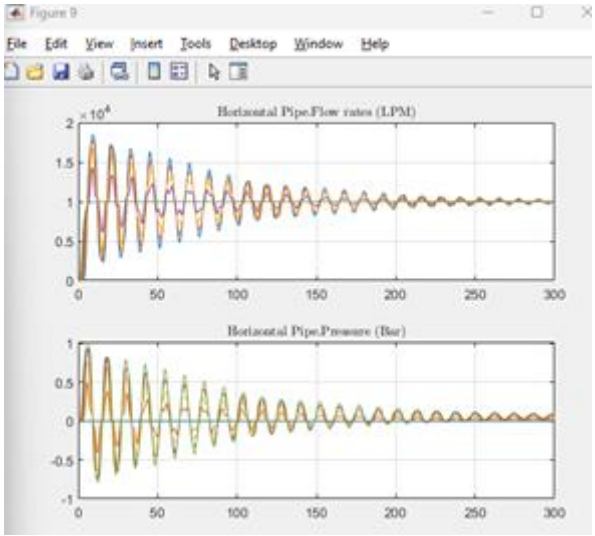

Figure 9

Figure 9 yields some unexpected results, but given the large annular gap, it's challenging to establish clear expectations. The parameters set for this case are far from typical drilling conditions, which makes it difficult to predict the behavior of the models. Both the Fjelde and Gjerstad models were designed to simulate more realistic scenarios, and their performance under these extreme conditions may not reflect their capabilities under normal drilling conditions.

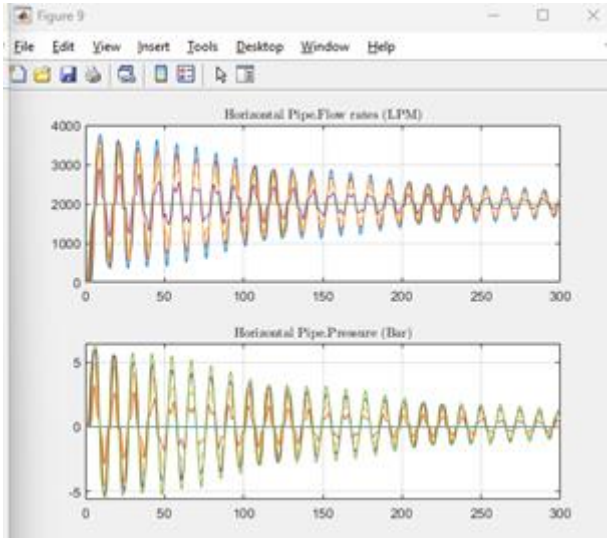

Figure 10

In the case of extremely low viscosity, both models again produce unusual results. The Gjerstad model, as shown in Figure 10, tends to oscillate in response to these edge-case conditions, while the Fjelde model, as shown in Figure 4, tends to yield a result of zero. This pattern suggests that the Gjerstad model may be more sensitive to extreme conditions, causing it to oscillate when faced with such low viscosity values. On the other hand, the Fjelde model appears to simplify the situation, resulting in a zero value.

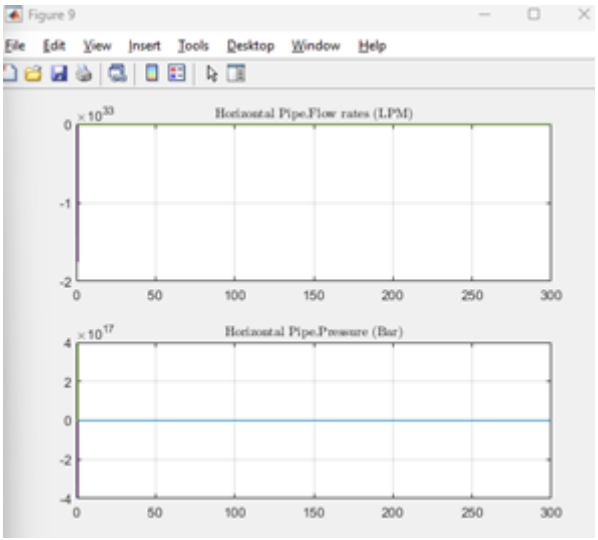

Figure 11

The small annular gap presents another unrealistic situation that both models struggle to handle accurately. The results from the Fjelde model, as shown in Figure 5, are markedly different from those of the Gjerstad model, as shown in Figure 11. This discrepancy could be due to a number of factors, but likely the Gjerstad model, being more complex, may be more sensitive to extreme conditions such as a small annular gap. This could cause it to produce results that differ significantly from the simpler Fjelde model.

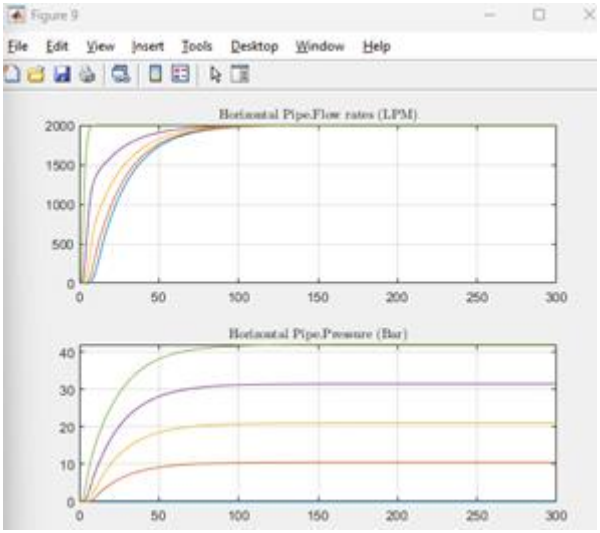

Figure 12

The final set of conditions presents a scenario that both models appear to handle well. As expected from the given conditions, the values displayed in Figure 12 from the Gjerstad model align closely with the pressure and ramp results from the Fjelde model, as shown in Figure 6. This agreement is particularly noteworthy given that the two models employ different methods for ramping up to their respective flower rates.

This suggests that both models are capable of accurately simulating more typical well conditions, despite their differences in complexity and calculation methods. It also underscores the importance of using realistic parameters in well simulations, as both models demonstrate better alignment and potentially higher accuracy under these conditions.

3.3 The efficiency and accuracy of the different shear stress calculations

3.3.1 Tw_NarrowSlotPowerLawLaminar.m

The Power Law case was implemented using the MATLAB function C.2 Tw_NarrowSlotPowerLawLaminar.m. The formula for the Power Law model is relatively simple, which makes it straightforward to implement and run. The average runtime for this case was approximately 2.5 seconds, which is slightly slower than the Newtonian case by about 0.6 seconds. Despite this, the Power Law case still demonstrates a high level of computational efficiency.

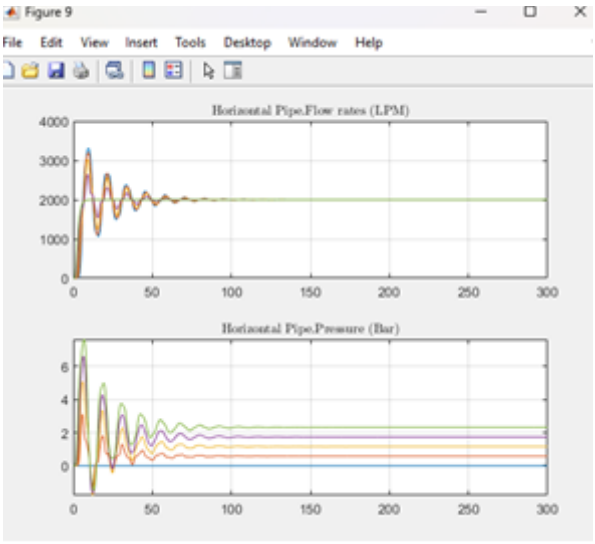

Figure 13 Power Law

However, it's worth noting that the results for the Power Law case, as shown in Figure 13, simulated using the values from Table 1, exhibit oscillating values that are quite different from the other cases. This is likely due to the values of the flow behaviour index (n) and the consistency index (k) in the Power Law model, which can represent a fluid that behaves very differently from a Newtonian fluid.

The calculated frictional pressure loss for the Power Law case, obtained using Equation 12 and the relationship in Equation 11, is approximately 2.32 bar. This value appears to be consistent with the plot in Figure 13, further validating the accuracy of the calculations for this case. Due

to the oscillations, it is difficult to tell if the Power Law model provides a reasonable approximation of the behaviour of non-Newtonian fluids under laminar flow conditions.

3.3.2 Tw_NarrowSlotBinghamPlasticLaminar.m

The Bingham Plastic model was implemented using the MATLAB function C.3 Tw_NarrowSlotBinghamPlasticLaminar.m. This case involves a more complex model due to the unique behavior of Bingham Plastic fluids, which exhibit a yield stress and behave like a solid until this yield stress is exceeded. The average runtime for this case was significantly longer than the other cases, at approximately 47 seconds. This is due to the additional complexity of the Bingham Plastic model, which requires more computational resources to accurately simulate.

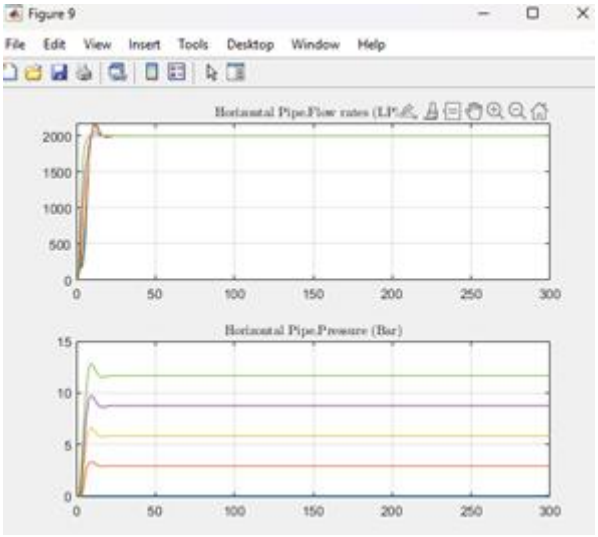

Figure 14 Bingham Plastic

The results for the Bingham Plastic case, as shown in Figure 14, indicate that the values at steady state are seemingly correct. By using Equation 13 and the relationship in Equation 11, we can calculate a value of approximately 11.6 bar at steady state, which aligns with the plot. However, it's important to note that the values while ramping the flowrate are likely inaccurate. This is because the current implementation of the Bingham Plastic model does not stop the flow when the yield stress is not overcome. This is a limitation of the current model and should be addressed in future work to improve the accuracy of the simulations.

3.3.3 Tw_NarrowSlotHerschelBulkley.m

The Herschel-Bulkley model, while more complex, was implemented with an average runtime of approximately 67 seconds. This is noticeably longer than the Bingham Plastic case, which had an average runtime of about 47 seconds. This increase in computational time is likely due to the more complex nature of the Herschel-Bulkley model, which includesadditional parameters, the flow behavior index (n) and the consistency index (k), in its calculations.

The Herschel-Bulkley model presents a unique aspect in that the equation used to calculate the wall shear stress (τ_w), as given by equation 14, is not always solvable. In such instances the function will result in imaginary values, which by default are ignored through the fsolve function, or the function resorts to using the simplified Bingham Plastic model, when there is no solution at all, to estimate τ_w. This situation introduces a degree of uncertainty over the model's accuracy, particularly during the ramping of the flow rate. This is because the use of the simplified Bingham Plastic model for cases where the wall shear stress (τ_w) is less than or equal to the yield stress (τ_y) could potentially introduce noise into the calculations.

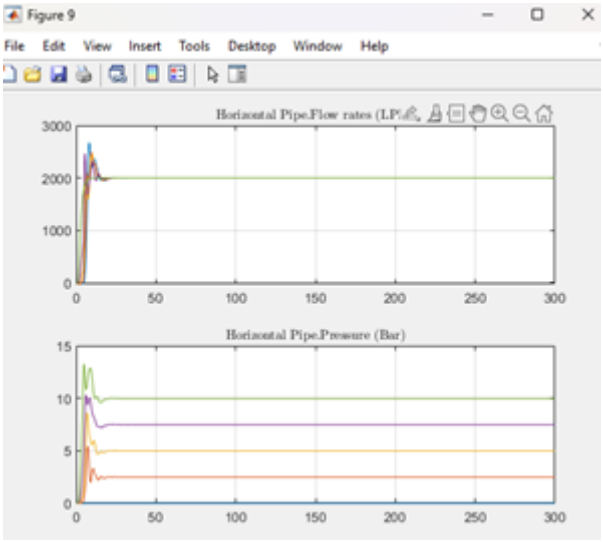

Figure 15 Herschel-Bulkley

Despite this, it's important to note that these uncertainties should resolve at steady state. As shown in Figure 15, the Herschel-Bulkley model reaches a steady state with a frictional pressure of approximately 10 bar. This value aligns well with calculations made using equation 14 for τ_w and equation 11 for the relationship between frictional pressure and τ_w, suggesting that

despite potential inaccuracies during the ramping phase, the model's accuracy at steady state remains robust.

3.4 Machine learning

3.4.1 Random Forest Regression

The machine learning model employed in this study is the Random Forest Regression model. This model is a type of ensemble learning method that constructs a multitude of decision trees during training and outputs the mean prediction of the individual trees. It is particularly effective in handling complex, non-linear relationships between variables, which makes it well-suited for predicting parameters in fluid dynamics and well simulations.

The Random Forest Regression model is based on calculations made by the C.5 Tw_NarrowSlotBinghamPlasticWithFlow.m script. The data was collected using C.6 TrainingDataCollector.m, and the model was trained using C.7 TrainingModel.m, resulting in the creation of C.8 Tw_NarrowSlotBPWithFlowML.m, to preload the model a line was added to A.1 MasterAlg_PipeHorizontal.m, see D.4 MasterAlg_PipeHorizontal.m.

This script estimates the narrow slot approximation for wall shear stress (τ_w) with the Bingham Plastic model under laminar flow conditions. The script also has the capability to set the flow rate out of a section to zero when the wall shear stress is less than or equal to the yield stress, which is a key feature of Bingham Plastic fluids. While training it to do this is possible, it would in this case be better to simply ad a function to return the signal to turn of flow if the estimated shear stress is less than the yield stress.

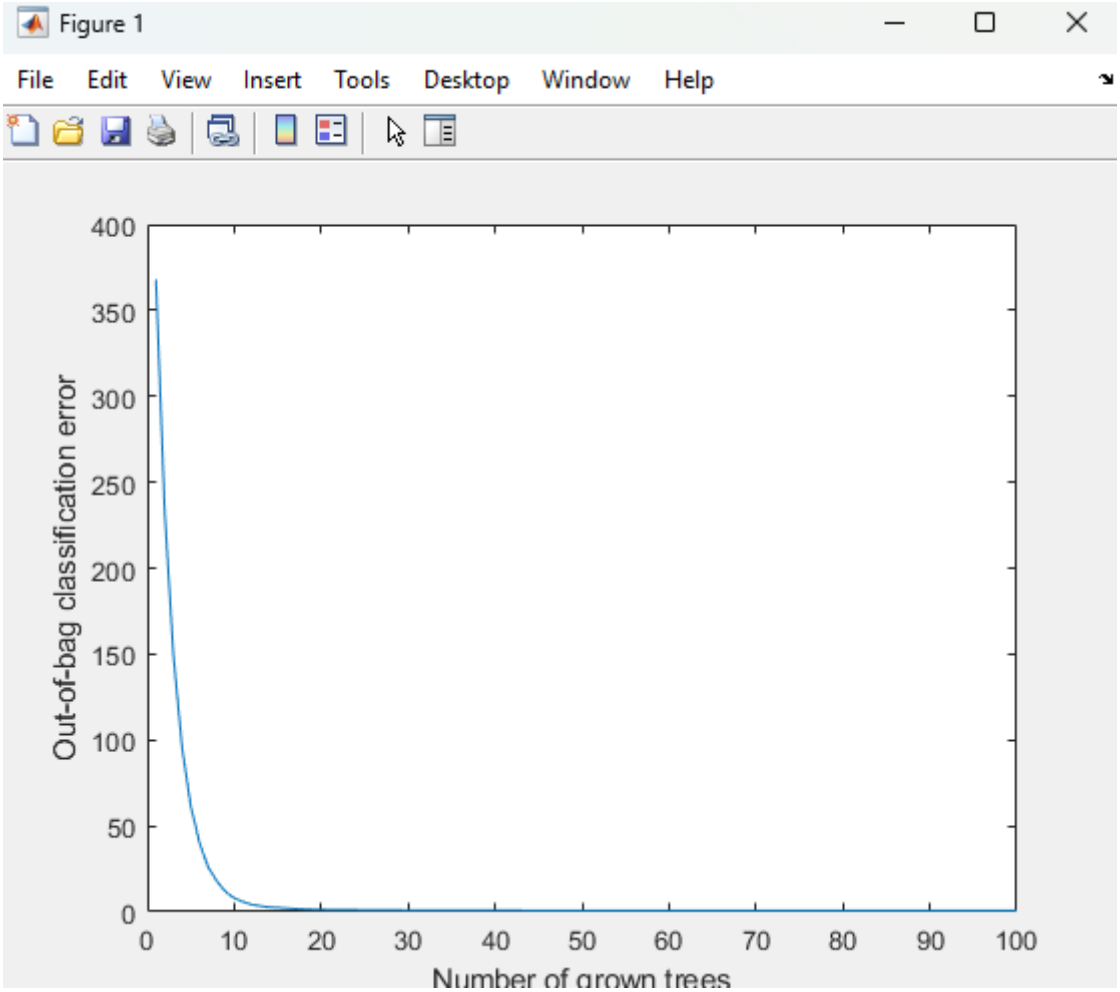

Figure 166 100 Grown trees

In the Random Forest Regression model, the number of trees grown, as shown in Figure 16, is a key parameter that can significantly influence the model's performance. Each tree in the forest is grown independently, and the final prediction is made by averaging the predictions of all the trees. Increasing the number of trees can improve the model's accuracy by reducing the variance of the predictions, but it also increases the computational cost and the time required to train the model. Conversely, reducing the number of trees can make the model faster to train and run, but it may also reduce the model's accuracy. Therefore, selecting the optimal number of trees is a crucial step in the model training process.

3.4.2 Root Mean Square Error

Now, let's consider the application of RMSE in the context of our machine learning model and the C.5 Tw_NarrowSlotBinghamPlasticWithFlow.m script.

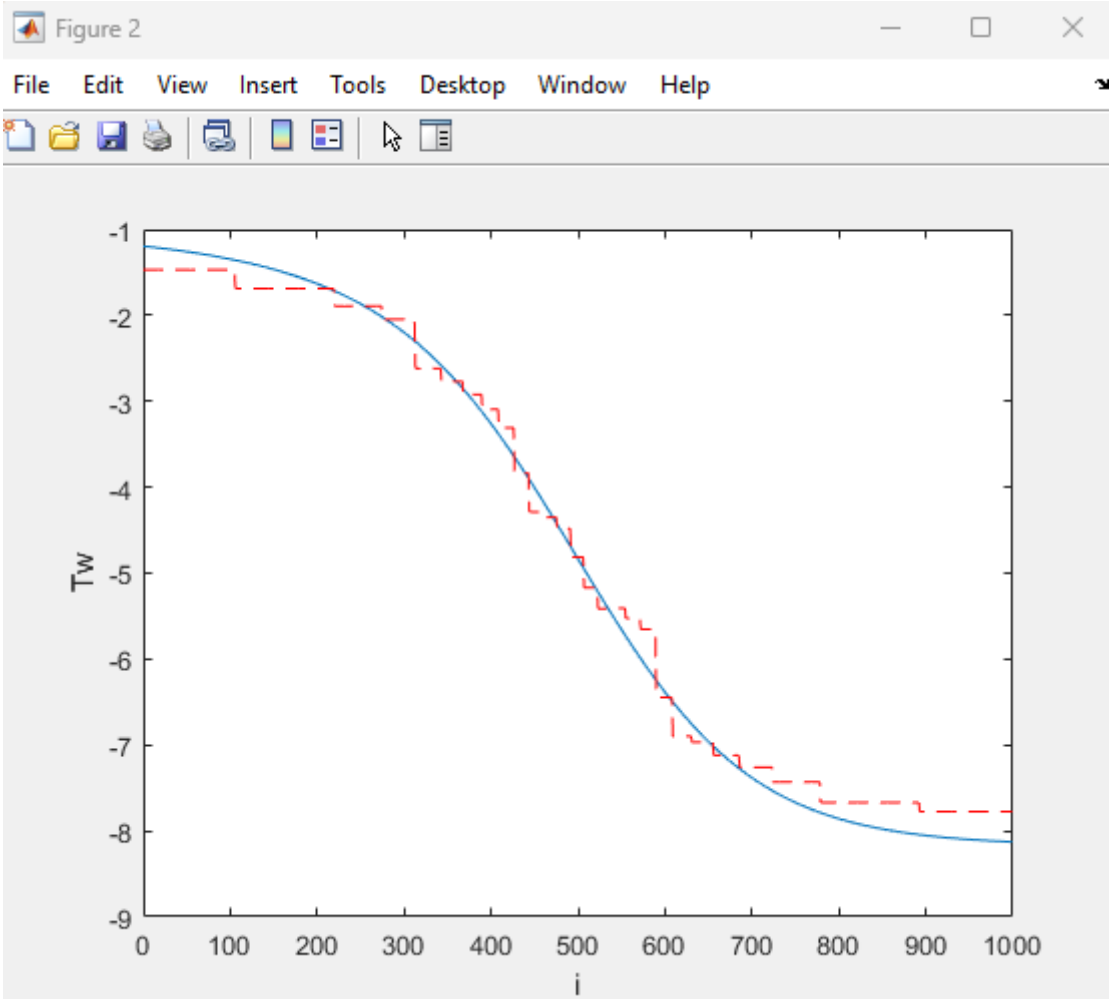

Figure 177 Visual comparison of calculation versus machine learning 100 trees

Figure 17 provides a visual comparison of the estimates made by the machine learning model and C.3 Tw_NarrowSlotBinghamPlasticLaminar.m when ran through the code designed for comparing, C.8 TestMLvsCalculations. The results from both methods are closely aligned, suggesting that the machine learning model is capable of accurately predicting the wall shear stress. Increasing the volume of data for training would likely enhance the accuracy of the machine learning model, and more closely fit the graphed curves.

However, visual comparison alone is not sufficient to determine the accuracy of the machine learning model. It's crucial to calculate the RMSE between the predicted and actual values to

quantify the model's performance. The RMSE measures the average magnitude of the errors in a set of predictions, without considering their direction. It aggregates the residuals (differences between predicted and actual values) into a single measure of predictive power. A lower RMSE indicates a better fit to the data.

By employing equation 15, as calculated in lines 37 to 40 in C.8 TestMLvsCalculations, we obtain a Root Mean Square Error (RMSE) of 0.2208. This value represents the standard deviation of the residuals, which are the prediction errors. Given that the range of the predicted values varies from -1 to -8.13, an RMSE of 0.2208 can be considered relatively low. This suggests that the machine learning model has a good fit to the data and is able to predict the wall shear stress with a reasonable level of accuracy. However, it's important to note that the acceptability of this RMSE value can be context-dependent and may vary based on specific application requirements.

3.4.3 Increasing Efficiency

In this particular case, reducing the number of trees in the Random Forest Regression model could be beneficial for improving computational efficiency. As shown in Figure 13, the model's accuracy does not significantly decrease when the number of trees is reduced. This suggests that a smaller forest could still provide reasonably accurate predictions while greatly reducing the computational cost and runtime.

Currently, running 1000 simulations with the Random Forest Regression model takes about 90 seconds on average, with 100 trees, which is significantly longer than the approximately 1 second required for 1000 runs of the C.3 Tw_NarrowSlotBinghamPlasticLaminar.m script. By reducing the number of trees in the Random Forest Regression model, it may be possible to bring the runtime closer to that of the C.3 script without sacrificing much in terms of accuracy.

In an effort to further optimize the computational efficiency of the Random Forest Regression model, the number of trees was reduced to 50. This adjustment resulted in a notable decrease in the average runtime for 1000 simulations, bringing it down to approximately 40 seconds. This is a significant improvement compared to the 90 seconds required when using 100 trees.

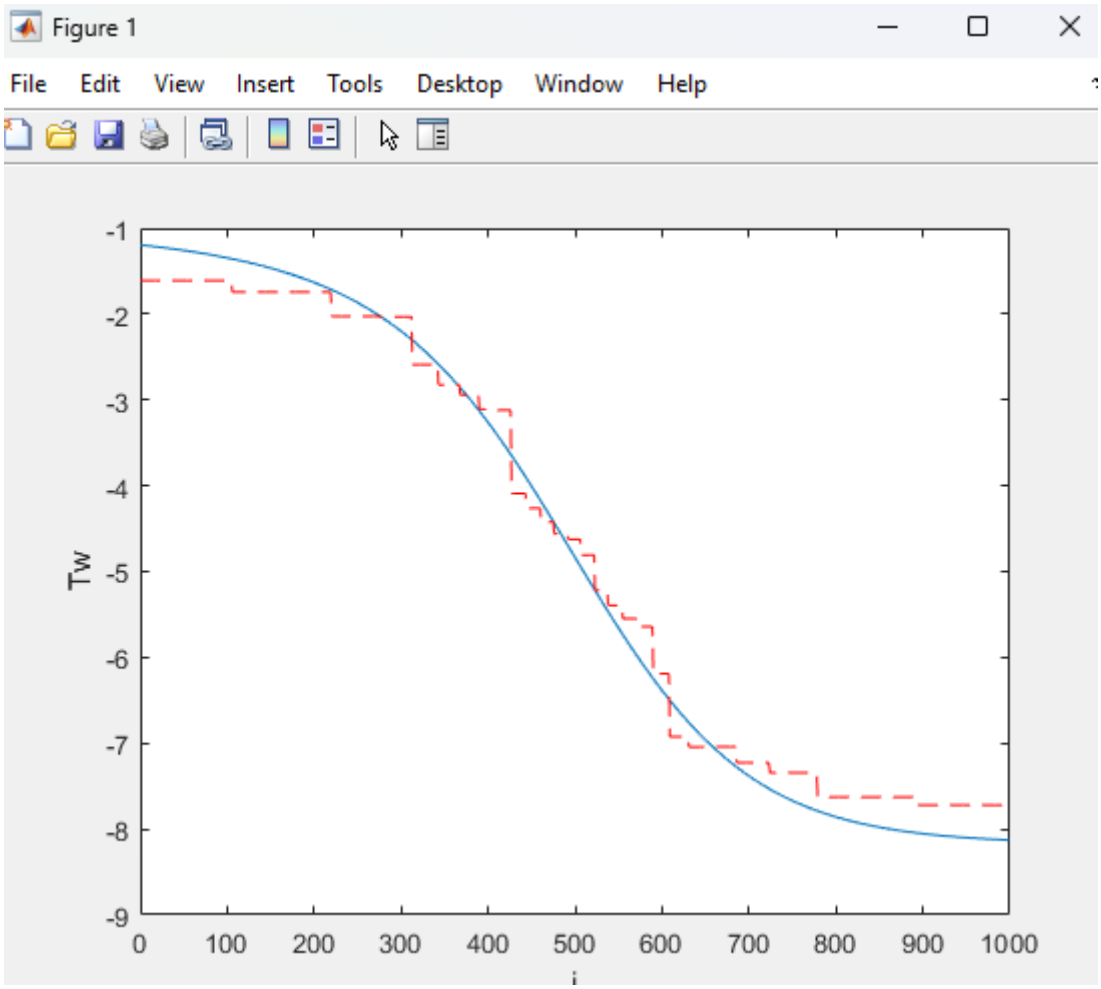

Figure 1818 Visual comparison of calculation versus machine learning 50 trees

Although visually hart to tell from Figure 18, it's important to note that while reducing the number of trees can decrease runtime, it may also affect the model's predictive accuracy. In this case, the Root Mean Square Error (RMSE) increased slightly to 0.2793, up from 0.2208 with 100 trees.

While this increase in RMSE is relatively small, it does indicate a slight decrease in the model's predictive accuracy. However, given the substantial reduction in runtime, this trade-off may be acceptable depending on the specific requirements of the simulation. In scenarios where speed is a priority, the slight decrease in accuracy might be considered negligible. Conversely, in situations where the highest possible accuracy is required, it might be preferable to use a larger number of trees, despite the longer runtime.

4 Conclusion

4.1 Comparison of the Models

The Gjerstad and Fjelde models are both effective tools for simulating drilling processes, each with their unique strengths and applications. The Gjerstad model, due to its design as an Ordinary Differential Equation (ODE) model, offers faster computational speed. This is largely due to the physics simplifications inherent in its design, which reduce the complexity of the calculations being executed, thereby enhancing computational efficiency.

Despite these simplifications, the Gjerstad model offers a more comprehensive framework for representing the drilling process. This robustness allows it to handle a wider variety of cases with accuracy, making it a valuable tool for scenarios where a more detailed and adaptable model is required.

On the other hand, the Fjelde model, being a Partial Differential Equation (PDE) model, provides a more detailed representation of the drilling process. While this results in increased computational time compared to the Gjerstad model, it also allows for a high level of accuracy in the simulation results.

Both models are flexible and can be tuned to handle most, if not all, physically possible cases. The simplicity of their codes allows for easy modifications, making them adaptable tools that can be adjusted to better suit specific needs.

4.2 Implementation of Functions for Non-Newtonian Fluids

The implementation of non-Newtonian fluid models presented varying levels of complexity and computational efficiency. The Power Law model was relatively straightforward to implement and provided accuracy comparable to the Newtonian model, albeit at a slightly slower computational speed.

On the other hand, the Bingham Plastic and Herschel-Bulkley models were more complex to implement and required significantly more computational resources, resulting in significantly

slower runtimes. While these models are more representative of real-world drilling fluids, their increased complexity presents challenges in terms of computational efficiency and accuracy.

In particular, the accuracy of the Herschel-Bulkley model during the ramp-up phase is difficult to evaluate due to the potential for the equation used to calculate wall shear stress (τ_w) to yield imaginary values. This introduces a degree of uncertainty into the calculations during this phase. However, it's important to note that these potential inaccuracies should resolve at steady state, as indicated by the model's performance under steady state conditions.

Overall, the choice of model should be guided by the specific requirements of the simulation, including the desired balance between computational speed, accuracy, and the complexity of the fluid behaviour to be simulated. It's also worth noting that more accurate solutions for the Bingham Plastic and Herschel-Bulkley models likely exist, and further research and development in this area could lead to improved simulation results.

4.2 Machine Learning

The implementation of the Random Forest Regression model in this study provided valuable insights into the potential of machine learning for simulating drilling processes. However, the results also highlighted some significant challenges.

The computational speed of the Random Forest Regression model was found to be considerably slower than the traditional models used in this study. Specifically, the runtime for the Random Forest Regression model was too long for practical applications, particularly for real-time use in drilling operations. This is a significant limitation, as one of the key requirements for a drilling simulation model is the ability to provide accurate results quickly enough to inform real-time decision-making.

One potential solution to this issue could be to replace the entire drilling simulation model with a machine learning model, rather than just the calculations for wall shear stress (τ_w). This would involve training a machine learning model to simulate the entire drilling process, rather than just a specific aspect of it. However, this would be a substantial undertaking, requiring a large amount of high-quality training data and significant computational resources.

5 References

Gjerstad, A. K. (2014). Simplified flow equations for single-phase non-Newtonian fluids in Couette-Poiseuille flow and in pipes : for dynamic modeling of surge and swab pressure in oil well drilling operations. *Universitetet i Stavanger Det teknisk-naturvitenskapelige fakultet*.

Saasen, A., Ytrehus, J., & Lund, B. (2020). Annular Frictional Pressure Losses for Drilling Fluids. *International Conference on Offshore Mechanics and Arctic Engineering*.

Wikipedia. (2023, June). *Machine Learning*. Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Machine_learning

Wikipedia. (2023, June). *Random Forest*. Hentet fra Wikipeia: https://en.wikipedia.org/wiki/Random_forest

Wikipedia. (2023, June). *Root-mean-square deviation*. Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Root-mean-square_deviation

Appendix A Alf Kristian Gjerstad Model

A.1 MasterAlg_PipeHorizontal.m

```
% ------------------------------------------------------------------
---
%
%               Main function
%
% Flow model of a horizontal pipe:
%   - divided into a number of volumes of choice
%   - based on ODE's, solved by own implementation of Runge-Kutta 4 /
Euler
%   - uses an explicit version of the Herschel-Bulkley rheology model
%
% ------------------------------------------------------------------
---


% Clearing memory:
clear

% Make space between console outputs
disp('...');

% Add paths where library files are located
NewLibraryPath = 'Library';
path(NewLibraryPath, path)
NewLibraryPath = '../MatlabLibraryKG';
path(NewLibraryPath, path)
%LibraryPath = 'd:/MatlabLibraryKG/FluidCalculations';
%path(LibraryPath, path)
%LibraryPath = 'd:/MatlabLibraryKG/InputsPreparationsAndPlotting';
%path(LibraryPath, path)


%% ----------- Setting Global constants -----------

% Physical Parameters:
GlobConstPhys = SetPhysicsParam();
GlobConstPhys = SetReynoldsNumberConstants(GlobConstPhys);
GlobConstPhys.dFlowInMaxPrSecCms = 0.1;     % Check



%% ---------- Simulation parameters -------------------
%GlobSimPar = SetSimParam(4880, 0.1, 'SimulationWellControlNickens');

SimulationTimeInit  = 100;              % in sec
GlobalTimestep      = 0.1; %0.05; 0.5
nGlobalSteps        = fix(SimulationTimeInit/GlobalTimestep);
SimulationTime      = GlobalTimestep * nGlobalSteps;
SimTimeVector       = 0:GlobalTimestep:SimulationTime-GlobalTimestep;

% Put into global structure
GlobalConstSim.GlobalTimeStep        = GlobalTimestep;
```

```
GlobalConstSim.nGlobalSteps          = nGlobalSteps;
GlobalConstSim.nInternalStepPrGlobal = 1;




%% ----------- Generating Input signals -----------
%
% STUDENT: This can be skipped if you want to generate your own time
series with input data

[FlowrateInput_LPM,             ThrottleInput_ClosePct]             =
PipeFluHrz_InputSignalGenerator(GlobalConstSim, GlobConstPhys);




%% -----------       Initiating the fluid      --------------------

% Fluid setup/selection - independent of ODE model     - NOMINAL values
FluidNom.Density                = 1500;
FluidNom.BulkModulus             = GlobConstPhys.BulkModulusObm;     %
or BulkModulusOil or BulkModulusWater
FluidNom.PresRefPa              = GlobConstPhys.PresAtmPa;

%FluidNom = SelectMudTypeOrSetParameters('HB1', FluidNom);
FluidNom  =  SelectMudTypeOrSetParameters('',  FluidNom,  0.2,  0.8,  0);
% Three last arguments are rheology parameters HB / PL


FluidNom.RheologyPipe                                                   =
ComputeRheologyParametersPipe(FluidNom.Viscosity.FlowBehaviorIndex); % Here
scalar input, but it may be a vector
% The SmoothFactor is independent of other variables => use the Nominal
value (% PUTTE SmoothFactor INN I ComputeRheologyParametersPipe)
FluidNom.RheologyPipe.SmoothFactor = +1;    % Valid range is [-2, +3],
where 0 is default, and optimal for moderate yp-values
%                                           % Neagtive factor may be
chosen to obtain sharper yp-effet (the treshold value is reduced by a decade
when the factor is reduced by one).
%                                         % Positive factor will make
the yp effect smoother and increase computational speed (P_T = P_T_default
.* 10.^SmoothFactor)



%% -----------    Activating object models and plotting   _----------

PipeFluHrzODE_InUse         = 1;
PlottingPipeFluHrzODE_InUse = 1;

PipeFluHrzPDE_InUse         = 0; % 1
PlottingPipeFluHrzPDE_InUse = 0; % 1

PlottingInputs_InUse = 1;
```

```
      %% ----------- Initiating the model PipeFluHrz --------------------

      InitBoundaries.PresBoundDnStmBar = GlobConstPhys.PresAtmBar * 10;    %
The pressure at the outlet
      InitBoundaries.FlowBoundUpStmLpm = 0;                                %
The flowrate at the inlet (but will be changed...?)


      % Set the Pipe properties FROM LAST cell to the FIRST cell
      % - In the Future, Read from GUI/Config file and Align to chosen Grid-
structure
      %
      SetGridPropertiesIndividually = 0;      % Select how to set these

      if SetGridPropertiesIndividually == 1
          Par.LengthPrGrid        = [1000, 800, 500, 800, 1000];
          Par.DiameterInnIn       = [4, 5, 5, 5, 4]; %[3, 4, 5, 6, 6];
          Par.ThrottleGridOrPoint = 1;
          Par.ConstrictionOpenPst = [10, 100, 100, 100, 100];
          Par.ThrottleActive      = [0, 1, 0, 0, 0];              % Activate
throttling - Might not work..
          Par.InclFromVrtDeg      = [90, 90, 90, 90, 90];
          Par.nGrids              = length(Par.LengthPrGrid);
      else
          LenTotal            = 3000;
          Par.nGrids          = 4; % 100
          UnityVectorGrids    = ones(1, Par.nGrids);
          Par.LengthPrGrid    = LenTotal/Par.nGrids * UnityVectorGrids;
          Par.DiameterInnIn       = 5 * UnityVectorGrids;     % Inches
          %Par.DiameterInnIn       = 0.1 * 100 *1/2.54 * UnityVectorGrids; %
m * m2cm * cm2In
          Par.ThrottleGridOrPoint = 1;
          Par.ConstrictionOpenPst = [100, 100*ones(1, Par.nGrids-1)];
          Par.ThrottleActive      = 0*UnityVectorGrids;
          %Par.ThrottleActive(2)   = 1;                            % Activate
throttling
          Par.InclFromVrtDeg      = 90 * UnityVectorGrids;
      end

      disp(['Length total: ', num2str(LenTotal)]);
      disp('...');



      % Call the Setup function, which Maps input units and forms to ODE
units and forms, Establish variables for plotting and Calls Init function
      if PipeFluHrzODE_InUse
          ObjectName = 'PipeWith1In2Out_Generic';
          [PipeFluHrzIntFc,                PipeFluHrzObj]                =
PipeFluHrz_2xOrd_Setup(ObjectName,     InitBoundaries,     Par,     FluidNom,
GlobConstPhys, GlobalConstSim);
      end
      if PipeFluHrzPDE_InUse
          ObjectName = 'PipeWith1In2Out_PDE1';
```

```
          [PipeFluHrzPdeIntFc,              PipeFluHrzPdeObj]              =
PipeFluHrz_SemiImplicitPde_Setup(ObjectName, InitBoundaries, Par, FluidNom,
GlobConstPhys, GlobalConstSim);
      end




      %%

      % ------------------------------------------------------------------
-----
      %   Main time loop executing the ODE models, calculating additional
      %   variables and putting all outputs into time vectors
      % ------------------------------------------------------------------
-----


      for t = 1:1:nGlobalSteps-1


          % --------------          Mapping inputs           --------------
---

          if PipeFluHrzODE_InUse
              PipeFluHrzIntFc.InputsPrev      = PipeFluHrzIntFc.Inputs;    %
Storing old values
              PipeFluHrzIntFc.Inputs.FlowUpStmLpm  = FlowrateInput_LPM(t);
              PipeFluHrzIntFc.Inputs.PresDnStmBar                          =
InitBoundaries.PresBoundDnStmBar;  %PressureAtmosphereBar;
              PipeFluHrzIntFc.Inputs.ThrottleClosePct                      =
ThrottleInput_ClosePct(t);
          end
          if PipeFluHrzPDE_InUse
              PipeFluHrzPdeIntFc.InputsPrev                                =
PipeFluHrzPdeIntFc.Inputs;    % Storing old values
              PipeFluHrzPdeIntFc.Inputs.FlowUpStmLpm                       =
FlowrateInput_LPM(t);
              PipeFluHrzPdeIntFc.Inputs.PresDnStmBar                       =
InitBoundaries.PresBoundDnStmBar;  %PressureAtmosphereBar;
              PipeFluHrzPdeIntFc.Inputs.ThrottleClosePct                   =
ThrottleInput_ClosePct(t);
          end




          % --------------            EXECUTING              --------------
---

          if PipeFluHrzODE_InUse  % EXECUTING:    PipeFluHrz:
              [Outputs,                   PipeFluHrzObj]                   =
PipeFluHrz_Step(PipeFluHrzIntFc.Inputs,         PipeFluHrzIntFc.InputsPrev,
PipeFluHrzObj, GlobConstPhys, GlobalConstSim, "ODE");
              % Note this is MAtlab's way to do "CALLED BY REFERENCE"
              OutputVector = [Outputs.Flow; Outputs.Pres; Outputs.Dens];
```

```
            PipeFluHrzIntFc.PlotMatrix(:,t+1) = OutputVector;
            PipeFluHrzIntFc.Outputs = Outputs;
        end

        if PipeFluHrzPDE_InUse  % EXECUTING:    PipeFluHrz:
            [Outputs,                PipeFluHrzPdeObj]                =
PipeFluHrz_Step(PipeFluHrzPdeIntFc.Inputs,    PipeFluHrzPdeIntFc.InputsPrev,
PipeFluHrzPdeObj, GlobConstPhys, GlobalConstSim, "PDE");
            % Note this is MAtlab's way to do "CALLED BY REFERENCE"
            OutputVector = [Outputs.Flow; Outputs.Pres; Outputs.Dens];
            PipeFluHrzPdeIntFc.PlotMatrix(:,t+1) = OutputVector;
            PipeFluHrzPdeIntFc.Outputs = Outputs;
        end    %



        % Show preogress by writing to console every 10 sec.
        SimTimeSec = t*GlobalTimestep;
        if (mod(SimTimeSec, 10) == 0)
            display(['Simulated time (sec) = ', num2str(SimTimeSec), ' of
', num2str(SimulationTimeInit)])
        end


    end


    disp('Simulation finised')



    % -------------------     Plotting      -------------------------

    if (PipeFluHrzODE_InUse || PipeFluHrzPDE_InUse)  %PipeFluHrz_InUse
        PlotMultiPrSub(9,    SimTimeVector,    PipeFluHrzIntFc.PlotMatrix,
PipeFluHrzIntFc.Par.OutputNames,             'Horizontal             Pipe',
PipeFluHrzIntFc.Par.PlotGrouping);   % Dll
    end


    if PlottingInputs_InUse == 1    % Collecting and plotting input signals:
        if (PipeFluHrzODE_InUse || PipeFluHrzPDE_InUse)  %PipeFluHrz_InUse
            InputVectors = [FlowrateInput_LPM'; ThrottleInput_ClosePct'];
            InputVariableNames = {'Flowrate'; 'ThrottlingInput'};
        else
            InputVectors              =              [FlowrateInput_LPM';
TravelBlockVelocityInput_0_1_ms'; ThrottleInput_ClosePct'];
            InputVariableNames   =   {'Flowrate';   'TravelBlockVelocity';
'ThrottlingInput'};
        end
        PlotSimple(1,  SimTimeVector,  InputVectors,  InputVariableNames,
'InputVariables');
    end
```

A.2 AlignVectorsValuesToMultipleLength.m

```
      % Aligns values of vector of one length to another vector length when
the
      % two vectors are multiples of each other


      function                          OutputVector                          =
AlignVectorsValuesToMultipleLength(InputVector, LenOutputVec)

          LenInputVec = length(InputVector);
          OutputVector = zeros(LenOutputVec, 1);

          if LenOutputVec < LenInputVec
              nInputsPrOutput = LenInputVec/LenOutputVec;
              OutputVectorSum = zeros(LenOutputVec, 1);  %nInputsPrOutput
              for i=1:LenOutputVec
                 for j=1:nInputsPrOutput
                     OutputVectorSum(i)     =     OutputVectorSum(i)     +
InputVector(2*(i-1) +j);
                 end
                 OutputVector(i) = OutputVectorSum(i)/nInputsPrOutput;
              end

          elseif LenOutputVec > LenInputVec
              nOutputsPrInput = LenOutputVec/LenInputVec;
              j = 1;
              for i=1:LenOutputVec
                  OutputVector(i) = InputVector(j);
                  if (i/j) == nOutputsPrInput
                      j=j+1;
                  end
              end
          else
              OutputVector = InputVector;
          end

          %OuputVector = OutputVector;
      end
```

A.3 CalcFluidDensityFromEqOfState.m

```
      % ------------------------------------------------------------------
-----
      %
      % Calculates the density of a Liquid from the Equation of state.
      % All numeric inputs may be vectors, but can also be scalars, or a mix.
      % If different fluid are present in a string or borehole, both Pressure,
      %  DensityNominalVec and BulkModulus can be vectors with individual
values
      % for each grid.
      %
      % ------------------------------------------------------------------
-----
```

```
      function    FluidDensity    =    CalcFluidDensityFromEqOfState(Pressure,
PressureReference, DensityNom, BulkModulus)

          PressureDifference = Pressure - PressureReference;
          FluidDensity    =    DensityNom    +    DensityNom./BulkModulus    .*
PressureDifference;

      end
```

A.4 ComputeRheologyParametersPipe.m

```
      % ------------------------------------------------------------------
------
      % Parameters for explicit Herschel-Bulkley rheology model - ref paper:
      % - Kristian Gjerstad, Rune W. Time, B. Erik Ydstie and Knut S.
Bjørkevoll.
      %   An Explicit and Continuously Differentiable Flow Equation for
      %    non-Newtonian Fluids in Pipes. SPE Journal 19 (1): 78-87. SPE-
165930-PA,
      %   2013.
      % ------------------------------------------------------------------
------


      function                          RheologyPipe                        =
ComputeRheologyParametersPipe(FlowBehaviorIndex)

          RheologyPipe.xi           =   0.97   -   0.1   *FlowBehaviorIndex   -
0.11*FlowBehaviorIndex.^2;
          RheologyPipe.sigma  = 0.20 + 0.45 *(FlowBehaviorIndex - 0.5).^2;
          RheologyPipe.psi    = 0.82 + 0.8 *FlowBehaviorIndex.^3;

          RheologyPipe.PT     = ( 1   + 45 .*(FlowBehaviorIndex+1).^(-5.4) )
* 0.0001;
          RheologyPipe.Omega  = 1 ./ FlowBehaviorIndex;
          RheologyPipe.SmoothFactor = 0;  % Valid range is [-2, +3], where 0
is default, and optimal for moderate yp-values
                                                % For high yp neagtive
factor may be chosen to obtain sharper yp-effet (for low flow rates).
                                                % For low yp, positive
factor will increase computational speed.
      end
```

A.5 fRampAndHold3.m

```
      %
**********************************************************************
      %
      % Function that returns a signal with one or several consecutive
```

```
      % step changes filtered through a filter (1. or 2. order).
      % The integral of the signal, is also returned.
      %
      %
************************************************************************


      % Improvement:
      % The functions return one row pr global timestep and one column pr
internal



      function        [SignalFinal,        SignalIntegrated]                =
fRampAndHold2(GlobalConstSim,    filterOrder,    filterTime,    startValue,
holdTimes, holdValues, ZeroLimit)

          GlobalTimeStep          = GlobalConstSim.GlobalTimeStep;
          nGlobalSteps            = GlobalConstSim.nGlobalSteps;
          nInternalStepPrGlobal   = GlobalConstSim.nInternalStepPrGlobal;

          T = GlobalTimeStep;
          simTimeTotal = GlobalTimeStep * nGlobalSteps;


          rampNumbers = length(holdValues);
          if rampNumbers ~= length(holdTimes)
              'The last two inputvectors must be of same length'
              pause
          end

          if sum(holdTimes) > simTimeTotal
              'Sum of holdTimes must be less than simTimeTotal'
              pause
          end


          Signal  = startValue * ones(1, fix(holdTimes(1)/T));            %
Signal values before first step
          for i = 2:rampNumbers                                           %
Loop through the numbers of steps for the total input signal
                  signalHold_i     =     holdValues(i-1)     *     ones(1,
fix(holdTimes(i)/T));         % Construct constant "hold-values" to be added
after the step
                  Signal = [Signal  signalHold_i];                        %
Combine the filtered step function with the constant hold-values for the
current step
          end
          restSamples = nGlobalSteps - length(Signal);
          Signal = [Signal   holdValues(rampNumbers) *ones(1, restSamples)];


          t               = 0;
          y  = Signal;

          if filterOrder == 0
              y(i) = Signal(i);               % No filter
```

```
        elseif filterOrder == 1
            alpha   =  GlobalTimeStep/(filterTime+GlobalTimeStep);
            for i=2:nGlobalSteps
                y(i) = (1-alpha) * y(i-1)   +   alpha*Signal(i);   % simple
low pass filter
                %t = [t, i*GlobalTimeStep];
            end
        elseif filterOrder == 2
            T  = GlobalTimeStep;
            w0 = 1/filterTime;      % Undamped resonance frequency
            % numerator:
            a1 = w0^2*T^2;
            a2 = 2*a1;
            a3 = a1;
            % denominator:
            f1 = (w0*T + 2);
            f2 = (w0*T - 2);
            b1 = f1^2;          % = 4 + 4*w0*T + w0^2*T^2;
            b2 = 2*f1*f2;       % = 2*w0^2*T^2 - 8;
            b3 = f2^2;          % = 4 - 4*w0*T + w0^2*T^2;
            %Parameters_tot = (a1+a2+a3-b2-b3)/b1;         % Always unity
for unity amplification

            for i=3:nGlobalSteps
                y(i) = 1/b1 * ( -b2*y(i-1) -b3*y(i-2) + a1*Signal(i) +
a2*Signal(i-1) + a3*Signal(i-2) );  % Critical damped 2. order filter (unity
amplification)
            end
        else
           'Filter type is missing'
        end

        for i=1:nGlobalSteps
            if (exist ('ZeroLimit', 'var')) == 1
                if y(i) < ZeroLimit
                    y(i) = 0;
                end
            end
        end

        SignalFinal         = y';
        SignalIntegrated     = cumtrapz(SignalFinal) * T;            % The
integral


    end
```

A.6 fReynoldsNumber.m

```
    %
*********************************************************************
    %
    % Returns an equivalent Reynolds number (Re) for non-Newtonian flow
    % in pipes and annulus. The Reynoldsnumber is always positive.
    %
```

```
      % To avoid problems of zero values in the denominator when the flow
rate
      % is zero, a "net Reynoldsnumber" where the flow velocity is taken out
is
      % also returned (this is ok since this flow velocity will cancel out
in
      % subsequent computations).
      %
      % The value of the input parameter G determines whether the output is
      % valid for pipe or annulus.
      %
      %
************************************************************************



      function [Re Re_net] = fReynoldsNumber(v_tot, k, n, density, h, G)

      % Input vectors:
          % v_tot     TOTAL effective velocity. For annulus it will be higher
than effective Couette-Poiseulille velocity when ws>0. CONSIDER USING TAYLOR
NUMBER.
          %           For pipes it is the axial bulk flow velocity regardless
of ws. CONSIDER INCLUDING AN EFFECT OF ROTATION INSIDE STRING IN THE FUTURE
          % k, n,     Rheolgy parameters
          % density   Density of fluid
          % h         Equals R_well-R_pipe for annulus, and equals the radius
for pipes
          % G            Constant that is equal (1/(3n+1)^n) for pipes and
(1/(4n+2)^n) for annulus


          Re      = 8 * density ./ k .* (h.*n).^n  .* G .* abs(v_tot).^(2-
n);
          Re_net  = 8 * density ./ k .* (h.*n).^n  .* G;


      end
```

A.7 fTransition.m

```
      %
************************************************************************
      %
      % Function that determines the transition from laminar to turbulent
flow
      % for PIPE and ANNULUS. Returns a transition number called FlowRegime
      % in the range [0 1].
      %
      % Output: FlowRegime =
      %       1           -> Laminar,
      %       0           -> Fully turbulent,
      %       <0 1>       -> Transitional
      %
      % The critical Reynoldsnumber and "center Reynoldsnumber" are dependent
on
      % the rheology parameter n.
```

```
      %
      % Rotation of inner cylinder will not affect this transition function.
      % However, for the annulus there will be a higher Reynoldsnumber used
as
      % input (computed in separate function), so that turbulence will start
      % earlier when the string rotates. For the pipe, string rotation does
not
      % affect the transition at all.
      %
      %
*************************************************************************


      function [FlowRegime, Re_cr, Re_ce] = fTransition(Re, n, Re_cr_nom,
Re_tu_nom, Re_Delta_n0, Trns_minVal)


      % Input vectors:
          % Re            Reynoldsnumber
          % n             Flow behavior index

      % Scalar input variables:
          % Re_cr_nom      = 2100;      Nominal value for critical Reynolds
numnber when transition region starts - effective value depends on rheology
          % Re_tu_nom     = 2900;     Nominal value for Reynolds numnber when
100% turbulence is achieved (at the center of the Transitional region) -
effective value depends on rheology
          % Re_Delta_n0    = 1370;      Addition in critical Reynoldsnumber
when n -> 0.
          % Trns_minVal    = [5 10]     Value in % of transition function at
critical Reynoldsnumber, Re_cr



          % Reynoldsnumber values at start and center point for transition
region:
          Re_ce_nom    = (Re_cr_nom + Re_tu_nom)/2;               % Nominal
center point
          Re_cr         = Re_cr_nom + (1-n) *Re_Delta_n0;         % Start of
transition region
          Re_ce         = Re_ce_nom + (1-n) *Re_Delta_n0;         % Current
center point for transition region


          % Compoutes the exponential power in the laminar-turbuelent
transition function
          %   - gives the "slope" defined by input parameters { log_base(x)
= log_10(x) / log_10(base) }
          GAM                     =    log10((100-Trns_minVal)/Trns_minVal)
./log10(Re_ce./Re_cr);


          % The smooth transition between laminar and turbulent flow:
          FlowRegime    = 1 ./ ( 1 + (Re ./ Re_ce).^GAM );


      end
```

A.8 NanAndInfCheck.m

```
function IsNan = NanAndInfCheck(InputVector, VariableName)

    IsNan = sum(isnan(InputVector));
    if (IsNan)
        disp(['A value in ', VariableName, ' is NaN'])
    end

    IsInf = sum(isinf(InputVector));
    if (IsInf)
        disp(['A value in ', VariableName, ' is Inf'])
    end

end
```

A.9 PipeFlu_PdeGen_Init.m

```
% ------------------------------------------------------------------
-----
%
% Building the Objects internal variables and parameters from prepared
inputs
%
%
% Initiating vaiables in a generic PDE-based object for the fluid inside
% a Pipe. Parameers must have been configured alreay.
%
% This function must be wrapped by an outer function that sends inn
%  LengthPrGridTd,  inclination,  density  and  initial  pressure  and
density.

% Hence, this functin is Generic since variants related to initial
% conditions, inclination, form of inputs/outputs etc are taken care
of
% by the outer wrapper.
%
% Geometry parameters and States are here set into a Structure to be
used
% by the PDEs.
%
% Preparing Outputs for input-mapping is not needed here since the
outer
% wrapper take care of this. Similarly; Other setup for plotting and
% PlotMatrix are not needed.
%
% The reference frame for the fluid is the moving solid pipe.
% Hence, the solid string acceleration gives a fictitious force.
%
% ------------------------------------------------------------------
-----

function PipeFluObj = PipeFlu_PdeGen_Init(ObjectName, p, FlowInitLpm,
PresInitBar, DensityInit, FluidNom)
```

```
            % Simulation constants:
            c.ObjectName    = ObjectName;
            c.T_max         = 0.01;             % Max Time step for PDE model
when using ??? solver


            % ---   Parameters   ---

            % Grid parameters
            p.nGridsDs       = length(p.LengthPrGrid);
            p.LengthTotal = sum(p.LengthPrGrid);

            % Grid-Vectors
            p.ZeroVectorGrids   = zeros(1, p.nGridsDs);
            p.UnityVectorGrids  = ones(1, p.nGridsDs);

            % Height
            dHeightVertical     = p.LengthPrGrid .* sin(p.InclHrzRad);
            p.HeightVerticalSum = sum(dHeightVertical);

            % Geometry vector parameters - These are constants during
simulation (unless Events are build):
            p.AreaCrs   = pi * p.RadiusInn.^2;
            p.AreaSrf   = pi * 2*p.RadiusInn .* p.LengthPrGrid;
            p.Volume    = p.AreaCrs .* p.LengthPrGrid;
            p.ConstrArea      = p.ConstrictionOpenPst./100 .* p.AreaCrs;   %
Additional constriction that can be used for Tool joint effect etc
            p.ConstrDischarge = 0.90;

            % Parameters that might be changes in Event Builder are stored in
an "Org"-parameter to be able to manipulate the original pr timestep and
restore it later
            p.RadiusInnOrg  = p.RadiusInn;
            p.AreaCrsOrg    = p.AreaCrs;




            % ---   Initial conditions for variables  ---
            FlowLpm     = FlowInitLpm';
            PresBar     = PresInitBar';

            % Establish total States and Output vectors:
            StateVariables    = [FlowLpm; PresBar];
            p.ModelOrder        = length(StateVariables(:,1));



            % Fluid properties:
            p.Fluid           =           SetFluidParametersPrGridDs(FluidNom,
p.UnityVectorGrids);


            % Combined parameters (fluid and solid):
            p.Fluid.BulkModulusEff = p.Fluid.BulkModulus;    % Effective Bulk
modulus/stiffness is now from NOMINAL fluid properties
```

```
                                                        %     LATER:     =
f(FluidNom.BulkModulus, BoundaryStiffness);
                                                        %               OR
p.FluidSolid.BulkModulusEffective         =         f(FluidNom.BulkModulus,
BoundaryStiffness);


          % --- Set the States and Outputs ---

          PipeFluObj.States.Flow   = FlowLpm;
          PipeFluObj.States.Pres   = PresBar;

          PipeFluObj.Par = p;
          PipeFluObj.Cst = c;

      end
```

A.10 PipeFluGen_2xOrd_Init.m

```
      %  ------------------------------------------------------------------
-----
      %
      % Initiating of generic PIPE object for the fluid inside it.
      %
      % This function must be wrapped by an outer function that sends inn
      %  LengthPrGridTd,  inclination,  density  and  initial  pressure  and
density.
      % Hence, this functin is Generic since variants related to initial
      % conditions, inclination, form of inputs/outputs etc are taken care
of
      % by the outer wrapper.
      %
      % Geometry parameters and States are here set into a Structure to be
used
      % by the ODEs.
      %
      % Preparing Outputs for input-mapping is not needed here since the
outer
      % wrapper take care of this. Similarly; Other setup for plotting and
      % PlotMatrix are not needed.
      %
      % The reference frame for the fluid is the moving solid pipe.
      % Hence, the solid string acceleration gives a fictitious force.
      %
      %  ------------------------------------------------------------------
-----

      function    PipeFluObj    =    PipeFluGen_2xOrd_Init(ObjectName,    p,
FlowInitLpm, PresInitBar, DensityInit, FluidNom)


          % Simulation constants:
          c.ObjectName    = ObjectName;
```

```
            c.T_max_RK4     = 0.025;            % Max Time step for ODE model
when using RK4 solver
            c.T_max_Euler   = 0.01;             % Max Time step for ODE model
when using Euler solver


            p.nGridsDs       = length(p.LengthPrGrid);
            %p.LengthPrGridDs = LengthPrGrid;
            p.LengthTotal = sum(p.LengthPrGrid);


            % Global constants:
            p.BiasFlowDs = 0;
            p.BiasPresDs = p.nGridsDs;
            p.BiasDensDs = p.nGridsDs*2;


            % Vectors:
            p.ZeroVectorGrids   = zeros(1, p.nGridsDs);
            p.UnityVectorGrids  = ones(1, p.nGridsDs);



            dHeightVertical     = p.LengthPrGrid .* sin(p.InclHrzRad);
            p.HeightVerticalSum = sum(dHeightVertical);



            % ---   Initial conditions   ---

            FlowLpm     = FlowInitLpm';
            PresBar     = PresInitBar';

            % Establish total States and Output vectors:
            StateVariables    = [FlowLpm; PresBar];
            p.ModelOrder        = length(StateVariables(:,1));

            % Output variables in addition to the States:
            %OutputVariables = [DensityInit'];
            %p.OutputOrder              =     length(StateVariables(:,1))     +
length(OutputVariables);
            %PipeFluObj.OutputsInitial = [StateVariables; OutputVariables];  %
Currently not in use, but maybe later




            % ---   String and Fluid properties/Boundaries   ---

            %  Geometry  vector  parameters  -  These  are  constants  during
simulaiton (unless Events are build):
            %p.RadiusInn = p.DiameterInn ./ 2;
            p.AreaCrs   = pi * p.RadiusInn.^2;
            p.AreaSrf   = pi * 2*p.RadiusInn .* p.LengthPrGrid;
            p.Volume    = p.AreaCrs .* p.LengthPrGrid;
```

```
            p.ConstrArea      = p.ConstrictionOpenPst./100  .* p.AreaCrs;   % Additional constriction that can be used for Tool joint effect etc
            p.ConstrDischarge = 0.90;

            % Parameters that might be changes in Event Builder are stored in an "Org"-parameter to be able to manipulate the original pr timestep and restore it later
            p.RadiusInnOrg  = p.RadiusInn;
            p.AreaCrsOrg    = p.AreaCrs;



            % Fluid properties:
            p.Fluid           =           SetFluidParametersPrGridDs(FluidNom, p.UnityVectorGrids);


            % Combined parameters (fluid and solid):
            p.Fluid.BulkModulusEff = p.Fluid.BulkModulus;    % Effective Bulk modulus/stiffness is now from NOMINAL fluid properties
                                                             %     LATER:     = f(FluidNom.BulkModulus, BoundaryStiffness);
                                                             %               OR p.FluidSolid.BulkModulusEffective         =         f(FluidNom.BulkModulus, BoundaryStiffness);


            % --- Set the States and Outputs ---

            PipeFluObj.States.Flow   = FlowLpm;
            PipeFluObj.States.Pres   = PresBar;

            PipeFluObj.Par = p;
            PipeFluObj.Cst = c;

        end
```

A.11 PlotMultiPrSub.m

```
        function EndCode = PlotMultiPrSub(fig, x, y, variableNames, ModelName, PlotGrouping)
        %xMin, xMax, yMin, yMax, Legends, XLab, YLab, xTickPos, xTickLab, xTickTxt, yTickPos, yTickTxt, yTickLab, fig, txtTitle, Log, LegBox, lineType, lineWidth, SubFig, LegPos)

            FntSz0 = 6+2+1;
            FntSz1 = 7+2+1;
            FntSz2 = 8+2+1;
            FntSz3 = 9+2+1;
            FntNm  = 'Times';

            figure(fig);
```

```
            TotalSubPlotNumber = length(PlotGrouping);
            graphsTotal = 0;
            for subWindow = 1:TotalSubPlotNumber
                subplot(TotalSubPlotNumber, 1, subWindow);
                hold off
                for graphsInSub=1:PlotGrouping(subWindow)
                    graphsTotal = graphsTotal + 1;
                    plot(x, y(graphsTotal,:))
                    txtTitle  =  [ModelName,  '.',  variableNames{subWindow}];
% 'State variables';
                    title(txtTitle,    'FontSize',FntSz0,    'FontName',FntNm,
'Interpreter','latex');
                    grid on
                    hold on
                end
            end
            EndCode = 1;
        end
```

A.12 PlotSimple.m

```
        function EndCode = PlotSimple(fig, x, y, variableNames, ModelName)
        %xMin,  xMax,  yMin,  yMax,  Legends,  XLab,  YLab,  xTickPos,  xTickLab,
xTickTxt, yTickPos, yTickTxt, yTickLab, fig, txtTitle, Log, LegBox, lineType,
lineWidth, SubFig, LegPos)

            FntSz0 = 6+2+1;
            FntSz1 = 7+2+1;
            FntSz2 = 8+2+1;
            FntSz3 = 9+2+1;
            FntNm  = 'Times';

            nVectors    = length(y(:,1));

            figure(fig);

            for i=1:nVectors            % Marching vertical through vectors
                yv = y(i,:);
                if length(x(:,1)) > 1
                    xv = x(i,:);
                else
                    xv = x(1,:);
                end
                subplot(nVectors, 1, i);
                plot(xv, yv)
                txtTitle = [ModelName, '.', variableNames{i}];      % 'State
variables';
                title(txtTitle,      'FontSize',FntSz0,      'FontName',FntNm,
'Interpreter','latex');
                grid
            end
```

```
        EndCode = 1;

    end
```

A.13 PressureDropOverConstriction.m

```
    % ------------------------------------------------------------------
-----
    %
    % Returns pressure drop over a constriction (valve, nozzle or other)
for
    % the FlowrateCms given as input. Can operate on scalar values or
vectors.
    %
    % Opening percentage of the constriction, ThrottlingOpenPct, is another
    % input. If it is a scalar, all values in AreaConstr will be reduced
by
    % this percentage. If it is avector, the reductions are individual.
    %
    % All variables are in SI units.
    %
    % Note: If the Flowrate is non-zero, a zero ThrottlingOpenPct will give
    % infinite DeltaPressureConstrictionPa. This is an impossible situation
and
    % the calling function must make sure that Flowrate goes to zero when
the
    % ThrottlingOpenPct approaches zero. For safety, we add an error-
handling
    % condition here.
    %
    %       RENAME TO: CalcDeltaPressureConstrictionPa
    %
    % ------------------------------------------------------------------
-----

    function                DeltaPressureConstrictionPa                =
PressureDropOverConstriction(FlowrateCms,  Density,  Discharge,  AreaConstr,
ThrottlingOpenPct)

        AreaConstrEffective = ThrottlingOpenPct./100 .* AreaConstr;

        AreaConstrMin     = 1E-4;
        if min(AreaConstrEffective) < AreaConstrMin
            AreaConstrEffective = min(AreaConstrEffective, AreaConstrMin);
% Operator works for both scalar and vector inputs, or one of each (=> only
those elements below minimum will be increased)
            'Error: ThrottlingOpenPct or AreaConstr is too low for the
given Flowrate'
        end

       %if min(ThrottlingOpenPct) < 50
       %    'stop'
       %end
```

```
        DeltaPressureConstrictionPa = sign(FlowrateCms) .* Density ./ (2
*Discharge.^2 .*AreaConstrEffective.^2) .* (FlowrateCms).^2;

    end
```

A.14 SelectMudTypeOrSetParameters.m

```
    %  ------------------------------------------------------------------
------
    %
    %  ------------------------------------------------------------------
------

    % FluidNom bør inneholdew alt dette:
    %                Density: 1173
    %            BulkModulus: 1.6600e+09
    %              PresRefPa: 101000
    %              Viscosity: [1×1 struct]
    %     ReynoldsNmbTrnsPar: [1×1 struct]
    %           RheologyPipe: [1×1 struct]

    % + LEGE TIL EQUIV.NEWTONIAN?


    function FluidNom = SelectMudTypeOrSetParameters(MudName, FluidNom,
ConsistencyIndex, FlowBehaviorIndex, YieldPoint) % + OBM/WBM?

        switch MudName

            case 'Mariner_12In'
                FluidNom.Viscosity.ConsistencyIndex     = 0.343;
                FluidNom.Viscosity.FlowBehaviorIndex    = 0.727;
                FluidNom.Viscosity.YieldPoint           = 12.0;

            case 'Mariner_09In'     % CHECK
                FluidNom.Viscosity.ConsistencyIndex     = 0.3;
                FluidNom.Viscosity.FlowBehaviorIndex    = 0.8;
                FluidNom.Viscosity.YieldPoint           = 8.0;

            case 'HB1'
                FluidNom.Viscosity.ConsistencyIndex     = 0.281;
                FluidNom.Viscosity.FlowBehaviorIndex    = 0.828;
                FluidNom.Viscosity.YieldPoint           = 10.0;

            case 'HB2'
                FluidNom.Viscosity.ConsistencyIndex     = 0.5;
                FluidNom.Viscosity.FlowBehaviorIndex    = 0.85;
                FluidNom.Viscosity.YieldPoint           = 10.0;

            case 'Newtonian1'
                FluidNom.Viscosity.ConsistencyIndex     = 0.1;
                FluidNom.Viscosity.FlowBehaviorIndex    = 1.0;
                FluidNom.Viscosity.YieldPoint           = 0.0;

            otherwise
```

```
                FluidNom.Viscosity.ConsistencyIndex    = ConsistencyIndex;
                FluidNom.Viscosity.FlowBehaviorIndex                    =
FlowBehaviorIndex;
                FluidNom.Viscosity.YieldPoint           = YieldPoint;

        end


    end
```

A.15 SetFluidParametersPrGridDs.m

```
    % Endre navn til: SetFluidParametersPrGridDs    (Ds på slutten)


    % Fluid parameters:
        % -nominal values pr grid - May include depth dependencies later
(TVD):
        % -pressure and temperature dependencies cannot be included here
since they are unknown at this stage

    function FluidParametersPrGrid = SetFluidParametersPrGridDs(FluidNom,
UnityVectorGridsDsCur)

        p.Fluid.DensityNom   = FluidNom.Density * UnityVectorGridsDsCur;
% Nominal because density is also computed as an additional variable
        p.Fluid.BulkModulus        =        FluidNom.BulkModulus        *
UnityVectorGridsDsCur;
        p.Fluid.PresRefPa   = FluidNom.PresRefPa * UnityVectorGridsDsCur;

        % Viscosity parameters:
        p.Fluid.Viscosity.ConsistencyIndex                                  =
FluidNom.Viscosity.ConsistencyIndex * UnityVectorGridsDsCur;    % Should be
dependent on depth (pressure)
        p.Fluid.Viscosity.FlowBehaviorIndex                                 =
FluidNom.Viscosity.FlowBehaviorIndex * UnityVectorGridsDsCur;    % Can be
assumed independent of depth
        p.Fluid.Viscosity.YieldPoint                                        =
FluidNom.Viscosity.YieldPoint * UnityVectorGridsDsCur;            % Can be
assumed independent of depth
        % For testing:
        %p.Fluid.Viscosity.YieldPoint           = UnityVectorGridsDsCur *
0;

        % For circular pipe geometry:

        p.Fluid.RheologyPipe                                                =
ComputeRheologyParametersPipe(p.Fluid.Viscosity.FlowBehaviorIndex);

        % The SmoothFactor is independent of other variables => use the
Nominal value:
        p.Fluid.RheologyPipe.SmoothFactor                                   =
FluidNom.RheologyPipe.SmoothFactor;
```

```
      %        p.Fluid.RheologyPipe.xi        = FluidNom.RheologyPipe.xi  *
UnityVectorGridsDsCur;
      %       p.Fluid.RheologyPipe.sigma   = FluidNom.RheologyPipe.sigma  *
UnityVectorGridsDsCur;
      %        p.Fluid.RheologyPipe.psi      = FluidNom.RheologyPipe.psi  *
UnityVectorGridsDsCur;
      %     p.Fluid.RheologyPipe.PT           = FluidNom.RheologyPipe.PT *
UnityVectorGridsDsCur;
      %     p.Fluid.RheologyPipe.Omega        = FluidNom.RheologyPipe.Omega
* UnityVectorGridsDsCur;
      %                            p.Fluid.RheologyPipe.SmoothFactor     =
FluidNom.RheologyPipe.SmoothFactor * UnityVectorGridsDsCur;

          FluidParametersPrGrid = p.Fluid;

      end
```

A.16 SetPhysicsParam.m

```
      %--------------------------------------------------------------------
------
      % Physical constants and unit conversion factors are put into a common
      % structure
      %--------------------------------------------------------------------
------

      function PhysicsParam = SetPhysicsParam()

          % Physical constants
          PhysicsParam.g                 = 9.81;
          PhysicsParam.PresAtmPa         = 101000;
          PhysicsParam.PresAtmBar        = 1.01;
          PhysicsParam.BulkModulusWater  = 2.2E9;
          PhysicsParam.BulkModulusOil    = 1.38E9;
          PhysicsParam.BulkModulusObm    = 1.66E9;
          PhysicsParam.GasConstIdeal      = 8.31446;   % Ideal gas constant
8.3144598(48) J mol?1 K?1[1]

          % Conversion factors
          PhysicsParam.Bar2Pa     = 100000;
          PhysicsParam.Pa2Bar     = 1/100000;
          PhysicsParam.Lpm2Cms    = 1/60000;
          PhysicsParam.Cms2Lpm    = 60000;
          PhysicsParam.Deg2Rad    = pi/180;
          PhysicsParam.Ft2m       = 0.3048;
          PhysicsParam.In2m       = 0.0254;
          PhysicsParam.Gal2L      = 3.7854;
          PhysicsParam.Bbl2m3     = 0.1590;
          PhysicsParam.Ibm2Kg     = 0.4536;

          % Constants defining various zero-limits
          PhysicsParam.ZeroLimit1 = 1E-3;
          PhysicsParam.ZeroLimit2 = 1E-6;
```

```
        PhysicsParam.ZeroLimit3 = 1E-9;
        PhysicsParam.ZeroLimit4 = 1E-12;

        PhysicsParam = orderfields(PhysicsParam);

    end
```

A.17 SetReynoldsNumberConstants.m

```
    %
%************************************************************************
    %
    % Set constants used for calculating frictional pressure loss for
    % turbulent flow and transition to turbulent flow - i.e., different
    % Reynolds number related constants
    %
    %
%************************************************************************


    function                    [GlobalConstPhysic]                    =
SetReynoldsNumberConstants(GlobalConstPhysic)

        GlobalConstPhysic.ReNmbCstPipe.CritNom      = 2100;  % For pipes,
Nominal value for critical Reynolds numnber when transition region starts -
effective value depends on rheology
        GlobalConstPhysic.ReNmbCstAnnu.CritNom      = 2100;  % For ANNULUS
        GlobalConstPhysic.ReNmbCstPipe.TurbNom      = 2900;  % For pipes,
Nominal value for Reynolds numnber when 100% turbulence is achieved, i.e. at
the end of the Transitional region (or center?) - effective value depends on
rheology
        GlobalConstPhysic.ReNmbCstAnnu.TurbNom      = 2900;  % For ANNULUS
        GlobalConstPhysic.ReNmbCstPipe.Delta_n0     = 1370;  % For pipes,
Addition in critical Reynoldsnumber when flow behavior index n -> 0 (Herschel-
Bulkley / Bingham).
        GlobalConstPhysic.ReNmbCstAnnu.Delta_n0     = 1370;  % For ANNULUS
        GlobalConstPhysic.ReNmbCstPipe.TrnsMinVal   = 10;    % Value in %
of transition function at critical Reynoldsnumber, Re_cr (typical values: 5
- 10)
        GlobalConstPhysic.ReNmbCstAnnu.TrnsMinVal   = 10;    % For ANNULUS

    end
```

A.18 Solver_RK4_New.m

```
    % ------------------------------------------------------------------
-----
    %
    % Runge-Kutta 4 solver for any ODE-model on correct form.
    %
```

```
      % NumericStabilizationMode is an optional input parameter that may be
set
      % to one in order to remove ocsillations and change of sign between
the
      % RK iterations. It is useful for models that are discontinous at zero
      % like a fluid with a yield point.
      % If such conditions occur, the values for RK-iteration 2-4
      % is set to zero instead of having sign opposite of the first iteration.
      %   Hence,   numerical   instability   around   zero   flow   will   be
reduced/removed.
      %
      % ------------------------------------------------------------------
-----


      function   x_nextLocal      =      Solver_RK4_New(Model_ODE,   Inputs,
InputsPrGrid,   x_current,   VarAdditional,   Parameters,   ConstGlobalPhysic,
T_sim, NumericStabilizationMode)

          %global GlobalTimestep
          %tStart       = 2;               % Value for first time iteration
(Integer)
          %ModelSetup   =   [ModelName   &   'Setup'];          %   ModelName   =
TankWithPipeOutlet (=>TankWithPipeOutlet_Setup)
          %[U,     xIC,     p,     MO,     T_sim]     =     ModelSetup();     %
=TankWithPipeOutlet_Setup();
          %xRK = xIC;       % Struktur bør være mulig her da..
          %ModelOrder  = length(x_current);
          %xRK       = zeros(ModelOrder,1);            % State vector whose
values are set for each iteration according to the RK-method


          if (exist ('NumericStabilizationMode', 'var')) == 0
              NumericStabilizationMode = 0;
          end

          zeroVector  = zeros(length(x_current),1);
          x_nextLocal = zeroVector;
          xRK         = zeroVector;
          f           = zeros(length(x_current),4);

          for r=1:4
              switch r
                  case 1
                      xRK(:) = x_current;
                  case 2
                      xRK(:) = x_current + T_sim/2 *f(:,1);
                  case 3
                      xRK(:) = x_current + T_sim/2 *f(:,2);
                  case 4
                      xRK(:) = x_current + T_sim *f(:,3);
              end

              f(:,r) = Model_ODE(Inputs, InputsPrGrid, xRK, VarAdditional,
Parameters, ConstGlobalPhysic);

          end

          % Evaluation of next time-step of the whole state vector x:
```

```
        for j=1:length(f(:,1))

            % if Solver == RK4:

            if NumericStabilizationMode == 1
                if x_current(j) < 1
                    if ( (f(j,1) * f(j,2) < 0) && (f(j,3) * f(j,4) < 0) &&
(f(j,1) * f(j,3) > 0) )
                        f(j,2) = 0;
                        f(j,4) = 0;
                        %nOscillatingNeglections = nOscillatingNeglections
+ 1;
                        %display(['RK4    is    neglecting    oscillating
derivatives for state variable ', num2str(j)])
                    elseif (f(j,1) * f(j,2) * f(j,3) * f(j,4)) < 0
                        f(j,2) = 0;
                        f(j,3) = 0;
                        f(j,4) = 0;
                        %nSignNeglections = nSignNeglections + 1;
                        %display(['RK4  is  neglecting  change  of  sign  of
derivatives for state variable ', num2str(j)])
                    end
                end
            end

            x_nextLocal(j) = x_current(j) + T_sim/6 * (f(j,1) +2*f(j,2)
+2*f(j,3) +f(j,4));


            if isnan(x_nextLocal(j))
                'Variable in xNext is NaN'
            end

        end

    end
```

A.19 TwPipeGeoLamTurb.m

```
    %
**************************************************************************
    %
    % The function returns average wall shear stress values (Tw) for non-
    % Newtonian fluids in circular PIPES when the bulk flow rate (VelLiq)
is the
    % dynamic input.
    % Outputs:
    %       Tw_HB_i:        Shear  stress  Herschel-Bulkley  (HB)  model,
combined/total.
    %   Tw_it:      Shear stress HB model, turbulent component.
    %   Tw_HB_il:   Shear stress HB model, laminar component.
    %   Tw_PL_il:   Shear stress Power-law (PL) model, laminar component.
    %
    % If the flow behavior index n = 1, Tw_HB_il reduce to laminar flow
for
    % Bingham plasic BP) fluids and Tw_PL_il reduces to laminar flow for
```

```
      % Newtonian fluids.
      %
      % Vectors where the elements represent each pipe segment/grid can be
      % provided as input => the output will be a vector.
      %
      % Tw is given explicitely as function of bulk flow rate, as opposed to
the
      % theoretical analytic equation, which is implicit and requires an
iterative
      % solution.

      % If SimMode == 1, the wall shear stress will be discontinuous at zero
flow.
      % In all other cases, the discontinuity at zero flow is handled by
smoothening.
      % For best accuracy, the SmoothFactor should be set to 0.
      % However, for non-zero yield points the function gets strongly non-
linear
      % around zero flow rate. This will slow down the ODE-solver if it is
set up
      % to give a predefined accuracy. Therefore, the smoothening function
can be
      % tuned to give faster respons by setting a value for the SmoothFactor
      % between zero and 1. The higher value, the faster response.
      %
      % Note:
      % The frictional pressure gradient (Pa/m) for HB fluids is given by:
      %   dp/dz_il =  2/RaIn * Tw_HB_il
      % The dimennsionless pressure gradient for HB fluids is given by:
      %   P_HB_il =  -Tw_HB_il / yp
      %
      %
      % by Kristian Gjerstad
      %
      %                   Version 1.0 - 2014.01.23
      %
      %
************************************************************************



      function          [Tw_HB_i,     Tw_it,     Tw_HB_il,     Tw_PL_il]     =
TwPipeGeoLamTurb(VelLiq,   Re_net,   FlowRegime,   GeometryPar,   FluidPar,
RheologyPar, SimMode)

      % Inputs (Note: Parameters here may be variables in calling functions):


          % Fluid:
          Dns          = FluidPar.Dns;              % Future: Bulk modulus,
etc
          ModelType    = RheologyPar.ModelType;     % Herschel Bulkley only
for current version
          k            = RheologyPar.ViscosityHb.k;
          n            = RheologyPar.ViscosityHb.n;
          yp           = RheologyPar.ViscosityHb.yp;

          xi           = RheologyPar.ContinuousSimplified.xi;
          sigma        = RheologyPar.ContinuousSimplified.sigma;
          psi          = RheologyPar.ContinuousSimplified.psi;
```

```
            P_T_default = RheologyPar.ContinuousSimplified.P_T_default;
            omega       = RheologyPar.ContinuousSimplified.omega;
            SmoothFactor = RheologyPar.ContinuousSimplified.SmoothFactor;

            % Geometry:
            RaIn        = GeometryPar.RaIn;
            %InclRad     = GeometryPar.InclRad;


            if (exist ('SimMode', 'var')) == 0
                SimMode = 0;
            end


        % Vector parameters:
            % k, n, yp,         Rheology parameters
            %   Dns                                              Fluid   density
* New
            % VelLiq,               Bulk flow velocity inside string RELATIVE
to the string velocity
            % RaIn,               Inner radius of the pipe
            % xi, sigma, psi,   Parameters used in f_yp
            % P_T_default, omega,   Parameters used in f_0
            %  Re_net,                The net Reynoldsnumber defined here as
Re/VelLiq^(2-n),
            %                       i.e. it is only dependent on the constant
parameters k, n, Dns and R

        % Scalar parameters:
            % FlowRegime         1 means 100% laminar, 0 means 100% turbulent
in-between means transitional flow
            % P_T_default,          A parameter in the smootening function for
laminar flow, designed to give best accuracy
            % SmoothFactor,     A factor in f_0, for increasing computational
speed of the ODE-solver
                              % Choosing 1 instead of 0 will slightly reduce
the accuracy around zero flow rates.


            % Laminar flow:
                [Tw_HB_il Tw_PL_il] = Tw_stringLaminar(k, n, yp, VelLiq, RaIn,
SmoothFactor, xi, sigma, psi, P_T_default, omega, SimMode);

            % Turbulent flow:
                Tw_it           = Tw_stringTurbulent(Re_net, n, VelLiq, Dns);


            % The combined solution:
                Tw_HB_i       = Tw_HB_il   .* (FlowRegime)   + Tw_it   .* (1-
FlowRegime);

        end



        function [Tw_HB_il, Tw_PL_il] = Tw_stringLaminar(k, n, yp, VelLiq,
RaIn, SmoothFactor, xi, sigma, psi, P_T_default, omega, SimMode)
```

```
      % Computing the shear stress for laminar flow (with dimensions):

          Tw_PL_il         =  -sign(VelLiq)  .*  k  .*(  (3*n+1)./(n.*RaIn)
.*abs(VelLiq) ).^n;                          % Shear stressfor PL fluids (i.e.
without the yield point)

          if yp == 0                  % When yp is 0, it reduces to PowerLaw
rheology. This condition is necessary to avoid dividing with zero when both
yp and VelLiq are zero.
              Tw_HB_il = Tw_PL_il;     % - although not neede for nonzero
VelLiq, it will be correct since PL and HB gives same results in this case.

          elseif SimMode == 1     % 100% YP-effect
              xi = 1;
              f_yp     = 1 + n./(2*n+1) .*(1 -xi.*(sigma.*yp ./(sigma.*yp +
abs(Tw_PL_il))).^psi);  % TODO: Denne er litt feil pga f0 kompensasjon - Finn
eksakt (enklere)
              Tw_HB_il  =  Tw_PL_il    +    sign(Tw_PL_il)  .*  yp  .*  f_yp;
% The complete function for HB fluids

          else
              f_yp     = 1 + n./(2*n+1) .*(1 -xi.*(sigma.*yp ./(sigma.*yp +
abs(Tw_PL_il))).^psi);     % The yield point effect of HB-fluids

              P_T                 =    P_T_default    .*    10.^SmoothFactor;
% SmoothFactor: Range: [-2, 3], Default: 0
              P_frac  = (abs(Tw_PL_il)./P_T).^omega;
              f_0     = sign(Tw_PL_il) .* P_frac ./ ( yp.^omega + P_frac );
% The smoothening function

              Tw_HB_il   =   (abs(Tw_PL_il)   +   yp   .*   f_yp)   .*   f_0;
% The complete function for HB fluids
          end

      end



      function Tw_it = Tw_stringTurbulent(Re_net, n, VelLiq, Dns)

      % Computing the shear stress for Turbulent flow (with dimensions):

          a        = (log10(n) +3.93)./50;      % Friction factor component
for non_Newtonian turbulent flow (Blasius-like approximation)
          b        = (1.75-log10(n))./7;        % Friction factor component
for non_Newtonian turbulent flow (Blasius-like approximation)


          Tw_it     = -1/2  *  a./(Re_net.^b)  .*  Dns  .*     sign(VelLiq)  .*
abs(VelLiq).^(2-2*b+n.*b);



      % Note, normally we would write:
          % Re     = 8*Dns/k *(2*RaIn*n/(6*n+2))^n *abs(VelLiq)^(2-n);     %
Reynoldsnumber
          % f     = a/(Re^b)                                          % friction
factor (Blasius)
```

```
          % Tw_it = -1/2 * f * Dns * VelLiq *abs(VelLiq);                  %
Wall shear stress

          % However, this will give infinite f for zero flow, and result in
Tw_it=NaN.
          % By rearranging and shortening the velocities, we get the correct
value Tw_it=0 for VelLiq=0.

      end
```

A.20 DsMain_Horizontal_2xOrd_ODE.m

```
      % ------------------------------------------------------------------
-----
      % ODE function for fluid inside the drillstring with 2 state variables
pr grid and nGr grids (2*nGr order ++)
      %
      % The drillstring is allowed to move and accelerate (velocityPipe >
0). In that case, the fluid inside the string will be
      % in a moving reference frame. As a consequence, the reference frame
will accelerate when the string is accelerating.
      % Hence, we have to include a 'fictitious' force of the fluid in the
drill string when the string is accelerating.

      % The string motion used here as input should ideally be taken from
the dynamic velocities for each solid element in a solid String model
      % However, only the velocity of the last element, giving bit velocity
and depth, will be used as input here.
      %
      % One of the inputs is ThrottleClosePct, which is for throttling the
flow area in the string or one
      % cell. This can be used for simulating scenarios like plugged drill
string.
      % ------------------------------------------------------------------
-----


      function  dxdt  =  DsMain_Horizontal_2xOrd_ODE(Inputs,  InputsPrGrid,
StateVector, Var, p, GlobCstPhysic)

          % Global Constants:
          g                   = GlobCstPhysic.g;
          Bar2Pascal          = GlobCstPhysic.Bar2Pa;
          Pascal2Bar          = GlobCstPhysic.Pa2Bar;
          LPM2CMS             = GlobCstPhysic.Lpm2Cms;
          CMS2LPM             = GlobCstPhysic.Cms2Lpm;

          nGr = p.nGridsDs;   % Number of cells/grids

          %ZeroFlowLimitLpm = 1E-34;
```

```
            % STUDENT: i HAVE Switch to LaminarOnly here. you must make the
missing code below
            %
            FlowRegime          = 'LaminarOnly';
            %FlowRegime          = 'LaminarTurb';


            % Input variables - Scalars:
            FlowUpStmLpm        = Inputs(1);
            PresDnStmBar        = Inputs(2);
            ThrottleClosePct    = Inputs(3);    % To be able to block/restrict
flow area in a grid (here scalar).

            % Input variables - Vectors:
            %AccDsAxlCurr      = -InputsPrGrid(:,3);    % Inverted sign since
fluid is positive down, while Solid string is positive up
            AccDsAxlCalc    = -InputsPrGrid(:,5);   % Inverted sign since fluid
is positive down, while Solid string is positive up
            %
            % Testing two different inputs for string accelerations:  TODO:
Remove 'AccDsAxlCurr' - Keep only the one calculated in DsFlu_Step => No
extra requirements on DsSol
            AccelerationPipe = AccDsAxlCalc';   % If switching to other input
- remember to also switch in BhaAndBit



            % Prepare vectors of State variables and density
            % (Extended vectors are vectors where an extra fictive cell is
added at front or at the end for easier looping)
            %
            FlowLpm = StateVector(p.BiasFlowDs+1:p.BiasFlowDs+nGr);
            FlowExtendedCms = [FlowLpm; FlowUpStmLpm]' * LPM2CMS;  % Adds the
top boundary flow at the end of the vector

            PresBar = StateVector(p.BiasPresDs+1:p.BiasPresDs+nGr);
            PresPaExtended  = [PresDnStmBar; PresBar]' * Bar2Pascal; % Adds the
downstream boundary flow at the beginning of the vector

            DensityExtended = p.Fluid.DensityNom(1) *ones(1, nGr+1);  % TODO:
Implement the function: Density = f(PressurePa)  % Inludes the dummy element

            % Calculate delta-flow and delta-pressure over the grids
            dFlowrateCms = p.ZeroVectorGridsDs;  % Flow in minus flow out for
each grid
            dPressurePa  = p.ZeroVectorGridsDs;    % Pressure in curret grid
minus pressure in downstream grid - for each grid
            for i=1:nGr
                dFlowrateCms(i)  = -FlowExtendedCms(i)  + FlowExtendedCms(i+1)
*DensityExtended(i+1)/DensityExtended(i);  % Note: Dummies are added
                dPressurePa(i) = PresPaExtended(i+1) - PresPaExtended(i);    %
Note: The indexes in PressurePaExtended are shifted one up due to dummy
            end

            % Remove extra element in the extended vectors
            FlowCms     = FlowExtendedCms(1:nGr);
            Density     = DensityExtended(1:nGr);
```

```
            % Option for adding drillstring wash-out / leakage etc.
            FlowLeakCms = p.ZeroVectorGridsDs;       % TODO: compute it from
annulus pressure and inputs



            %  ############  The  differential  equation  (ODE)  for  pressure
(Conservation of Mass) ############
            %
            % Rate of change of Pressure in Drill string:
            dDtPressureBar = Pascal2Bar *  p.Fluid.BulkModulus ./ p.Volume .*
(dFlowrateCms - FlowLeakCms);
            %
            %
#########################################################################
##################




            % Calculate the extra pressure drop due to throttling the a cell
or whole pipe:
            %
            if p.ThrottleGridOrPoint == 0
                % Throttle only one cell by p.ThrottleActive
                ThrottleOpenPctVct   =      100   *   p.UnityVectorGridsDs   -
(p.ThrottleActive .* ThrottleClosePct);
            else
                % Throttle the whole pipe
                ThrottleOpenPctVct   =      100   *   p.UnityVectorGridsDs   -
ThrottleClosePct;
            end
            dPresConstrAbsPa = PressureDropOverConstriction(FlowCms, Density,
p.ConstrDischarge,  p.ConstrArea,  ThrottleOpenPctVct);    % Positive since
FlowCms always > 0
            dPresConstrPa    = sign(FlowCms) .* dPresConstrAbsPa;

            % Calculate the pressure forces
            dPresEff = dPressurePa - dPresConstrPa;      % Effective pressure
over each grid
            Fp = p.AreaCrs .* dPresEff;                  % The pressure force
over each grid



            % Gravity forces:
            Fg  =  g  *  p.Fluid.DensityNom  .*  p.LengthPrGrid  .*  p.AreaCrs  .*
sin(p.InclHrzRad); % TODO: Make and Call function that calculates current
Density based on current pressure



            % Fictitious force if the string is accelerating (since the pipe
is the reference frame):
            MassFluid   = p.Volume .* p.Fluid.DensityNom;
            Ffic        = -MassFluid .* AccelerationPipe;    % Fictitious force
has  opposite  sign  as  acceleration  of  reference  frame  (pipe)  when
signconvention is equal
```

```
        % Minor forces:     % Future: Add forces resulting from tool joints
and other constrictions in the string/BHA
        Fmin = p.ZeroVectorGridsDs; % = Kmina1 .* (velocity_fluid.^2);


        % Wall shear stress for use in Friction forces:
        %
        if FlowRegime == 'LaminarOnly'  % STUDENT: USE THIS ONE

            VelocityPipe = 0;    % Since the pipe motion is the reference
frame
            GeometryPar.RadiusInn = p.RadiusInn;
            GeometryPar.AreaCrs   = p.AreaCrs;

            % STUDENT: FINISH THIS FUNCTION
            wallshearStressPipeLaminar                                    =
Tw_stringNewtonianLaminar(FlowCms, VelocityPipe, GeometryPar, p.Fluid);

            Ff = wallshearStressPipeLaminar .* p.AreaSrf;



        elseif FlowRegime == 'LaminarTurb'  % STUDENT: DONT'USE THIS ONE

            VelocityPipe = 0;    % Since the pipe motion is the reference
frame
            k  = p.Fluid.Viscosity.ConsistencyIndex;
            n  = p.Fluid.Viscosity.FlowBehaviorIndex;
            yp = p.Fluid.Viscosity.YieldPoint;

            % For Turbulent vs Laminar flow: Computing Reynoldsnumber and
weight-number for flow regime:
            G_empty      = (1./(3*n+1)).^n;          % Parameter valid for
empty hole below bit
            G            = G_empty;

            VelocityLiquid = FlowCms ./ p.AreaCrs - VelocityPipe;
            [ReLam  ReLamNet]  =  fReynoldsNumber(VelocityLiquid,  k,  n,
p.Fluid.DensityNom,  p.RadiusInn,  G);      % TODO:  p.Fluid.DensityNom  =
p.Fluid.DensityCur
            [FlowRegime,   Re_cr,   Re_ce]   =   fTransition(ReLam,   n,
GlobCstPhysic.ReNmbCstPipe.CritNom,      GlobCstPhysic.ReNmbCstPipe.TurbNom,
GlobCstPhysic.ReNmbCstPipe.Delta_n0, ...

GlobCstPhysic.ReNmbCstPipe.TrnsMinVal);   % TODO: Splitte i to funksjoner:
fReCriticalNCenter og fTransition (Returnerer samme Re_cr og Re_ce hver gang)
            Re_net       = ReLamNet;

            FluidPar.Dns             = p.Fluid.DensityNom;          % TODO:
Use p.Fluid.DensityCur;
            RheologyPar.ModelType    = 'Herschel Bulkley';
            RheologyPar.ViscosityHb.k   = k;
            RheologyPar.ViscosityHb.n   = n;
            RheologyPar.ViscosityHb.yp  = yp;
            RheologyPar.ContinuousSimplified.xi                           =
p.Fluid.RheologyPipe.xi;  % RheologyPipe is comuted for all grids initially
```

```
            RheologyPar.ContinuousSimplified.sigma                          =
p.Fluid.RheologyPipe.sigma;
            RheologyPar.ContinuousSimplified.psi                            =
p.Fluid.RheologyPipe.psi;
            RheologyPar.ContinuousSimplified.P_T_default                    =
p.Fluid.RheologyPipe.PT;
            RheologyPar.ContinuousSimplified.omega                          =
p.Fluid.RheologyPipe.Omega;
            RheologyPar.ContinuousSimplified.SmoothFactor                   =
p.Fluid.RheologyPipe.SmoothFactor;
            GeometryPar.RaIn = p.RadiusInn;          %GeometryPar.AreaCrs
= p.AreaCrs;

            SimMode = 1;    % Gives absolute YP
            [wallshearStressPipeTotal,   TwTurb,   TwLam,   TwPlLam]   =
TwPipeGeoLamTurb(VelocityLiquid, Re_net, FlowRegime, GeometryPar, FluidPar,
RheologyPar, SimMode);


            % Ensuring that the yp-force does not initiate flow when it
comes to rest (due to dicrete time points)
            %
            Ff = p.ZeroVectorGridsDs;
            for i = 1:nGr
                if abs(FlowLpm(i)) == 0     % OK to test against 0 here
because it i is set to zero
                    Fsum = Fp(i) + Fg(i) + Ffic(i) + Fmin(i);
                    if abs(wallshearStressPipeTotal(i)) > 0         % Will
never happen
                        'abs(wallshearStressPipeTotal(i)) > 0 - Will never
happen'
                    else
                        if yp(i) * p.AreaSrf(i) > abs(Fsum)      % If the
yp has higher potensial than Fsum, it is reduced to Fsum
                            Ff(i) = -Fsum;                          % Ff must
be set directly here to avoid numerical round-off
                            wallshearStressPipeTotal(i)                 =
Ff(i)/p.AreaSrf(i);
                            %wallshearStressPipeTotal(i)        =        -
Fsum/p.AreaSrf(i);
                        else
                            wallshearStressPipeTotal(i)    =    -yp(i)    *
sign(Fsum);
                            Ff(i)    =    wallshearStressPipeTotal(i)    .*
p.AreaSrf(i);
                        end
                    end
                else
                    Ff(i) = wallshearStressPipeTotal(i) .* p.AreaSrf(i);
                end
            end

        else
            'Flow regime type is missing'
            pause
        end
```

```
          % ############ Conservation of Momentum on fluid in Drill string
###########
          %
          % The derivative dQdt in SI units:
          dDtFlowLpm = CMS2LPM * p.AreaCrs ./ MassFluid .* (Fp + Ff + Fg +
Ffic + Fmin);  % Sign of Ff (Tw) is negative when flow>0
          %
          %
#########################################################################



          % ####### Aligning the outputs into one vector that is returned
#######
          %
          dxdt = [dDtFlowLpm'; dDtPressureBar'];
          %
          %
#########################################################################

      end
```

A.21 DsMain_Horizontal_2xOrd_OLD_ODE.m

```
      % ------------------------------------------------------------------
-----
      %
      % ODE function for fluid inside the drillstring with 2 state variables
pr grid and nGr grids (2*nGr order ++)
      %
      % The boundaries between the grids inside string can be different from
borehole grids, and their lengths may be varying.
      % All grid boundaries follow the string motion, including the first
one,
      % which may start at the hook or just below the Seabed (follows the
hook or StringSolid.PosBot(1)).
      %
      % The reference frame of the fluid is the moving grids. As a
consequence, the reference frame will accelerate when the string is
accelerating.
      % Hence, we have to include a 'fictitious' force of the fluid in the
drill string when the string is accelerating.

      % The string motion used here as input should ideally be taken from
the dynamic velocities for each solid element in StringSolid.
      % However, only the velocity of the last element, giving bit velocity
and depth, will be used here.
      %
      % The input ThrottlingOpenPct is for throttling the flow area in the
BHA
      % (assuming there is additional resistance due to mud motor or similar)
      % Scenarios for plugges drill string should be simulated by directly
      % manipulating the AreaConstr for relevant grids.
      %
```

```
      % ------------------------------------------------------------------
-----


      function  dxdt  =  DsMain_Horizontal_2xOrd_ODE(Inputs,  InputsPrGrid,
StateVector, Var, p, GlobCstPhysic)

          % Global Constants:
          g                   = GlobCstPhysic.g;
          %PresAtmospherePa    = GlobalConstPhysic.PresAtmPa;   % TODO: For
use later when pipe is disconnected
          Bar2Pascal          = GlobCstPhysic.Bar2Pa;
          Pascal2Bar          = GlobCstPhysic.Pa2Bar;
          LPM2CMS             = GlobCstPhysic.Lpm2Cms;
          CMS2LPM             = GlobCstPhysic.Cms2Lpm;

          %ZeroFlowLimitLpm = GlobCstPhysic.ZeroFlowLimitLpm;
          %ZeroFlowLimitLpm = p.ZeroFlowLimitLpm;
          ZeroFlowLimitLpm = 1E-34;

          %p.AreaSrf
          %MassFluid   = p.Volume .* p.Fluid.DensityNom;
          %p.Fluid.Viscosity.YieldPoint;



          %FlowRegime          = 'LaminarOnly';
          FlowRegime          = 'LaminarTurb';


          % Input variables - Scalars:
          FlowUpStmLpm        = Inputs(1);
          PresDnStmBar        = Inputs(2);
          ThrottleClosePct    = Inputs(3);    % To be able to block/restrict
flow area in a grid (here scalar).

          % Input variables - Vectors:
          %VelDsAxlCurr   = InputsPrGrid(:,1);
          %VelDsAngCurr   = InputsPrGrid(:,2);
          AccDsAxlCurr    = -InputsPrGrid(:,3);   % Inverted sign since fluid
is positive down, while Solid string is positive up
          %AccDsAngCurr   = InputsPrGrid(:,4);
          AccDsAxlCalc    = -InputsPrGrid(:,5);   % Inverted sign since fluid
is positive down, while Solid string is positive up

          % Testing two different inputs for string accelerations:  TODO:
Remove 'AccDsAxlCurr' - Keep only the one calculated in DsFlu_Step => No
extra requirements on DsSol
          %AccelerationPipe = AccDsAxlCurr';
          AccelerationPipe = AccDsAxlCalc';   % If switching to other input
- remember to also switch in BhaAndBit



          % Parameters:
          nGr = p.nGridsDs;   % = nBit = GridNumberBit;

          FlowLpm = StateVector(p.BiasFlowDs+1:p.BiasFlowDs+nGr);
```

```
%     if max(FlowLpm) > 1
%        'stop';
%     end
%for i = 1:nGr
%    if abs(FlowLpm(i)) > 0 && abs(FlowLpm(i)) < ZeroFlowLimitLpm
%        'Flow under Limit'
%    end
%end
FlowLpm_Test = FlowLpm .* (abs(FlowLpm) > ZeroFlowLimitLpm);
```

```
FlowExtendedCms = [FlowLpm; FlowUpStmLpm]' * LPM2CMS;  % Adds the top boundary flow at the end of the vector
PresPaExtended                          =           [PresDnStmBar; StateVector(p.BiasPresDs+1:p.BiasPresDs+nGr)]'  *  Bar2Pascal;  % Adds the downstream boundary flow at the beginning of the vector
DensityExtended = p.Fluid.DensityNom(1) *ones(1, nGr+1);  % TODO: Implement the function: Density = f(PressurePa)  % Inludes the dummy element

FlowLeakCms = p.ZeroVectorGridsDs;  % For drillstring wash-out etc -  TODO: compute it from annulus pressure and inputs

dFlowrateCms = p.ZeroVectorGridsDs;  % Flow in minus flow out for each grid
dPressurePa  = p.ZeroVectorGridsDs;    % Pressure in curret grid minus pressure in down stream grid - for each grid
for i=1:nGr
    dFlowrateCms(i)  = -FlowExtendedCms(i)  + FlowExtendedCms(i+1) *DensityExtended(i+1)/DensityExtended(i);  % Note: Dummies are added
    dPressurePa(i) = PresPaExtended(i+1) - PresPaExtended(i);    % Note: The indexes in PressurePaExtended are shifted one up due to dummy
end

FlowCms     = FlowExtendedCms(1:nGr);
%PressurePa  = PressurePaExtended(1:nGr);
Density     = DensityExtended(1:nGr);
```

```
%    ############    Conservation of Mass on fluid in Drill string ############

% Rate of change of Pressure in Drill string:
dDtPressureBar = Pascal2Bar *  p.Fluid.BulkModulus ./ p.Volume .* (dFlowrateCms - FlowLeakCms);  % There is no volume change here due to moving grids
```

```
%    ############    Conservation of Momentum on fluid in Drill string ############
```

```
        % Pressure forces:

        if p.ThrottleGridOrPoint == 0
            % Throttle at a point given by p.ThrottleActive
            ThrottleOpenPctVct        =    100  *  p.UnityVectorGridsDs  -
(p.ThrottleActive .* ThrottleClosePct);
        else
            % Throttle whole grid
            ThrottleOpenPctVct        =    100  *  p.UnityVectorGridsDs  -
ThrottleClosePct;
        end

        dPresConstrAbsPa = PressureDropOverConstriction(FlowCms, Density,
p.ConstrDischarge,  p.ConstrArea,  ThrottleOpenPctVct);    % Positive since
FlowCms always > 0
        dPresConstrPa    = sign(FlowCms) .* dPresConstrAbsPa;
    %     if min(dPresConstrPa) < 0
    %        'dPressureConstrictionPa < 0';
    %     elseif min(dPresConstrPa) > 0
    %         dPresConstrPa
    %     end
        Fp  =  p.AreaCrs  .*  (dPressurePa  -  dPresConstrPa);    % Effective
pressure over the grid

        % Gravity forces:
        Fg  =  g  *  p.Fluid.DensityNom  .*  p.LengthPrGrid  .*  p.AreaCrs  .*
sin(p.InclHrzRad); % TODO: Make and Call function that calculates current
Density based on current pressure

        % Fictitious force (since the pipe is the reference frame):
        MassFluid   = p.Volume .* p.Fluid.DensityNom;
        Ffic        = -MassFluid .* AccelerationPipe;    % Fictitious force
has  opposite  sign  as  acceleration  of  reference  frame  (pipe)  when
signconvention is equal

        % Minor forces:     % Future: Add forces resulting from tool joints
and other constrictions in the string/BHA
        Fmin = p.ZeroVectorGridsDs; % = Kmina1 .* (velocity_fluid.^2);




        % Wall shear stress for use in Friction forces:

        if FlowRegime == 'LaminarOnly'

            VelocityPipe = 0;    % Since the pipe motion is the reference
frame
            GeometryPar.RadiusInn = p.RadiusInn;
            GeometryPar.AreaCrs   = p.AreaCrs;
            [wallshearStressPipeLaminar,           Tw_PL_lam]           =
Tw_stringLaminarHBSimple(FlowCms, VelocityPipe, GeometryPar, p.Fluid);
            wallshearStressPipeTotal = wallshearStressPipeLaminar
```

```
            %Ff = wallshearStressPipeLaminar .* p.AreaSrf;

      %     elseif FlowRegime == 'LaminarOnly_OLD'
      %
      %         VelocityPipe = 0;   % Since the pipe motion is the reference
frame
      %         GeometryPar.RadiusInn = p.RadiusInn;
      %         GeometryPar.AreaCrs   = p.AreaCrs;
      %                         [wallshearStressPipeLaminar,   Tw_PL_lam]   =
Tw_stringLaminarHBSimple(FlowExtendedCms(1:nGr), VelocityPipe, GeometryPar,
p.Fluid);
      %         Ff = wallshearStressPipeLaminar .* p.AreaSrf;


      % TA MED DETTE:
            %[TwHbTot,       TwHbTurb,       TwHbLam,       TwPlLam]      =
TwPipeGeoLamTurb(VelLiq,   Re_net,   FlowRegime,   GeometryPar,   FluidPar,
RheologyPar)
            %TwReal = TwHbTot, YpReal, Time, Q)       % For å gjøre den
glatt men likevel med 100% YP-effekt med valgfri verdi ved Q->0.
            %Ff = TwHbTot .* p.AreaSrf;

         elseif FlowRegime == 'LaminarTurb'

            VelocityPipe = 0;   % Since the pipe motion is the reference
frame
            k  = p.Fluid.Viscosity.ConsistencyIndex;
            n  = p.Fluid.Viscosity.FlowBehaviorIndex;
            yp = p.Fluid.Viscosity.YieldPoint;

            % For Turbulent vs Laminar flow: Computing Reynoldsnumber and
weight-number for flow regime:
            %Ga          = (1./(4*na+2)).^na;        % Parameter valid for
annulus
            G_empty     = (1./(3*n+1)).^n;          % Parameter valid for
empty hole below bit
            %G           = [Ga(1:nBit);  G_empty(nBit+1:nBh)];
            G           = G_empty;

            VelocityLiquid = FlowCms ./ p.AreaCrs - VelocityPipe;
            [ReLam  ReLamNet]  =  fReynoldsNumber(VelocityLiquid,  k,  n,
p.Fluid.DensityNom,  p.RadiusInn,  G);      % TODO:  p.Fluid.DensityNom  =
p.Fluid.DensityCur
            [FlowRegime,   Re_cr,   Re_ce]   =   fTransition(ReLam,   n,
GlobCstPhysic.ReNmbCstPipe.CritNom,     GlobCstPhysic.ReNmbCstPipe.TurbNom,
GlobCstPhysic.ReNmbCstPipe.Delta_n0, ...

GlobCstPhysic.ReNmbCstPipe.TrnsMinVal);   % TODO: Splitte i to funksjoner:
fReCriticalNCenter og fTransition (Returnerer samme Re_cr og Re_ce hver gang)
            Re_net      = ReLamNet;

            FluidPar.Dns            = p.Fluid.DensityNom;         % TODO:
Use p.Fluid.DensityCur;
            RheologyPar.ModelType   = 'Herschel Bulkley';
            RheologyPar.ViscosityHb.k   = k;
            RheologyPar.ViscosityHb.n   = n;
            RheologyPar.ViscosityHb.yp  = yp;
            RheologyPar.ContinuousSimplified.xi                           =
p.Fluid.RheologyPipe.xi;  % RheologyPipe is comuted for all grids initially
```

```
            RheologyPar.ContinuousSimplified.sigma                    =
p.Fluid.RheologyPipe.sigma;
            RheologyPar.ContinuousSimplified.psi                      =
p.Fluid.RheologyPipe.psi;
            RheologyPar.ContinuousSimplified.P_T_default              =
p.Fluid.RheologyPipe.PT;
            RheologyPar.ContinuousSimplified.omega                    =
p.Fluid.RheologyPipe.Omega;
            RheologyPar.ContinuousSimplified.SmoothFactor             =
p.Fluid.RheologyPipe.SmoothFactor;
            GeometryPar.RaIn = p.RadiusInn;          %GeometryPar.AreaCrs
= p.AreaCrs;

            SimMode = 1;    % Gives absolute YP
            [wallshearStressPipeTotal,   TwTurb,   TwLam,   TwPlLam]   =
TwPipeGeoLamTurb(VelocityLiquid, Re_net, FlowRegime, GeometryPar, FluidPar,
RheologyPar, SimMode);


        else
            'Flow regime type is missing'
            pause
        end



        % Ensuring that the yp-force does not initiate flow when it comes
to rest (due to dicrete time points)
        Ff = p.ZeroVectorGridsDs;
        for i = 1:nGr
            %if abs(FlowLpm(i)) < ZeroFlowLimitLpm    % There is no flow
            %    if abs(FlowLpm(i)) > 0          % Just checking
            %        'stop'
            %    end

            if abs(FlowLpm(i)) == 0    % OK to test against 0 here because
it i is set to zero

                Fsum = Fp(i) + Fg(i) + Ffic(i) + Fmin(i);

                if abs(wallshearStressPipeTotal(i)) > 0           % Will
never happen
                    'abs(wallshearStressPipeTotal(i)) > 0 - Will never
happen'
                else
                    if yp(i) * p.AreaSrf(i) > abs(Fsum)      % If the yp
has higher potensial than Fsum, it is reduced to Fsum
                        Ff(i) = -Fsum;                       % Ff must be
set directly here to avoid numerical round-off
                        wallshearStressPipeTotal(i) =  Ff(i)/p.AreaSrf(i);
                        %wallshearStressPipeTotal(i) = -Fsum/p.AreaSrf(i);

                    else
                        wallshearStressPipeTotal(i) = -yp(i) * sign(Fsum);
                        Ff(i)     =     wallshearStressPipeTotal(i)     .*
p.AreaSrf(i);
                    end
                end
```

```
                    %if sign(Fsum) ~= sign(wallshearStressPipeTotal)  % burde
hatt (i)
                    %    if abs(wallshearStressPipeTotal(i) * p.AreaSrf(i)) >
abs(Fsum)
                    %        wallshearStressPipeTotal(i) = -Fsum/p.AreaSrf(i);
                    %    end
                    %end
                                %Fsum = abs(Fp(i) + Fg(i) + Ffic(i) + Fmin(i));
% All forces except friction
                                %if yp(i) * p.AreaSrf(i) > Fsum  % Sjekk at
Ffic skal være med her...
                    %                   if abs(wallshearStressPipeTotal(i)) *
p.AreaSrf(i) > Fsum
                    %                       wallshearStressPipeTotal(i) = -
Fsum/p.AreaSrf(i);
                    %             end

                else
                    Ff(i) = wallshearStressPipeTotal(i) .* p.AreaSrf(i);
                end
            end
            %Ff = wallshearStressPipeTotal .* p.AreaSrf;


            % The derivative dQdt in SI units:
            dDtFlowLpm = CMS2LPM * p.AreaCrs ./ MassFluid .* (Fp + Ff + Fg +
Ffic + Fmin);  % Sign of Ff (Tw) is negative when flow>0




        %   ############   Aligning the outputs    ############

            dxdt = [dDtFlowLpm'; dDtPressureBar'];


        end
```

A.22 PipeFlu_SemiImplicitPDE.m

```
        % ------------------------------------------------------------------
-----
        %
        % PDE function for fluid inside the drillstring with X variables pr
grid and nGr grids
        %
        % All grid boundaries follow the string motion, including the first
one,
        % which may start at the hook or just below the Seabed (follows the
hook or StringSolid.PosBot(1)).
        %
        % The reference frame of the fluid is the moving grids. As a
consequence, the reference frame will accelerate when the string is
accelerating.
```

```
      % Hence, we have to include a 'fictitious' force of the fluid in the
drill string when the string is accelerating.

      % The string motion used here as input should ideally be taken from
the dynamic velocities for each solid element in StringSolid.
      % However, only the velocity of the last element, giving bit velocity
and depth, will be used here.
      %
      % The input ThrottlingOpenPct is for throttling the flow area in the
BHA
      % (assuming there is additional resistance due to mud motor or similar)
      % Scenarios for plugged drill string should be simulated by directly
      % manipulating the AreaConstr for relevant grids.
      %
      % ------------------------------------------------------------------
-----


      function dxdt = PipeFlu_SemiImplicitPDE(Inputs, InputsPrGrid, VelLiq,
VelGas, Pres, Var, p, GlobCstPhysic, TsLocal)
                                                          %(Inputs,
InputsPrGrid, StateVector, Var, p, GlobCstPhysic)

          % Global Constants:
          g                   = GlobCstPhysic.g;
          Bar2Pascal          = GlobCstPhysic.Bar2Pa;
          Pascal2Bar          = GlobCstPhysic.Pa2Bar;
          LPM2CMS             = GlobCstPhysic.Lpm2Cms;
          CMS2LPM             = GlobCstPhysic.Cms2Lpm;

          ZeroFlowLimitLpm = 1E-34;

          %FlowRegime          = 'LaminarOnly';
          FlowRegime          = 'LaminarTurb';


          % Input variables - Scalars:
          FlowUpStmLpm        = Inputs(1);
          FlowUpStmGasLpm     = 0;
          PresDnStmBar        = Inputs(2);
          ThrottleClosePct    = Inputs(3);    % To be able to block/restrict
flow area in a grid (here scalar).

          % Input variables - Vectors:
          %VelDsAxlCurr   = InputsPrGrid(:,1);
          %VelDsAngCurr   = InputsPrGrid(:,2);
          AccDsAxlCurr    = -InputsPrGrid(:,3);   % Inverted sign since fluid
is positive down, while Solid string is positive up
          %AccDsAngCurr   = InputsPrGrid(:,4);
          AccDsAxlCalc    = -InputsPrGrid(:,5);   % Inverted sign since fluid
is positive down, while Solid string is positive up

          % Testing two different inputs for string accelerations:  TODO:
Remove 'AccDsAxlCurr' - Keep only the one calculated in DsFlu_Step => No
extra requirements on DsSol
          %AccelerationPipe = AccDsAxlCurr';
          AccelerationPipe = AccDsAxlCalc';    % If switching to other input
- remember to also switch in BhaAndBit
```

```
            % Parameters:
            nGr = p.nGridsDs;    % = nBit = GridNumberBit;

            FlowLpm = VelLiq .* p.AreaCrs;
            % Pres = StateVector(p.BiasPresDs+1:p.BiasPresDs+nGr)
            %, VelGas, Pres


            FlowExtendedCms = [FlowLpm; FlowUpStmLpm]' * LPM2CMS;  % Adds the
top boundary flow at the end of the vector
            PresPaExtended  = [PresDnStmBar; Pres]' * Bar2Pascal; % Adds the
downstream boundary flow at the beginning of the vector
            DensityExtended = p.Fluid.DensityNom(1) *ones(1, nGr+1);  % TODO:
Implement the function: Density = f(PressurePa)  % Inludes the dummy element

            FlowLeakCms = p.ZeroVectorGridsDs;   % For drillstring wash-out etc
-  TODO: compute it from annulus pressure and inputs

            dFlowrateCms = p.ZeroVectorGridsDs;  % Flow in minus flow out for
each grid
            dPressurePa  = p.ZeroVectorGridsDs;    % Pressure in curret grid
minus pressure in down stream grid - for each grid
            for i=1:nGr
                dFlowrateCms(i)  = -FlowExtendedCms(i) + FlowExtendedCms(i+1)
*DensityExtended(i+1)/DensityExtended(i);  % Note: Dummies are added
                dPressurePa(i) = PresPaExtended(i+1) - PresPaExtended(i);    %
Note: The indexes in PressurePaExtended are shifted one up due to dummy
            end

            FlowCms     = FlowExtendedCms(1:nGr);
            %PressurePa  = PressurePaExtended(1:nGr);
            Density     = DensityExtended(1:nGr);




        %    ############    Conservation of Mass on fluid in Drill string
############

            % Rate of change of Pressure in Drill string:
            dDtPressureBar = Pascal2Bar *  p.Fluid.BulkModulus ./ p.Volume .*
(dFlowrateCms - FlowLeakCms);  % There is no volume change here due to moving
grids




        %    ############    Conservation of Momentum on fluid in Drill string
############

            % Pressure forces:

            if p.ThrottleGridOrPoint == 0
                % Throttle at a point given by p.ThrottleActive
```

```
            ThrottleOpenPctVct        =    100  *  p.UnityVectorGridsDs  -
(p.ThrottleActive .* ThrottleClosePct);
        else
            % Throttle whole grid
            ThrottleOpenPctVct        =    100  *  p.UnityVectorGridsDs  -
ThrottleClosePct;
        end
        dPresConstrAbsPa = PressureDropOverConstriction(FlowCms, Density,
p.ConstrDischarge,  p.ConstrArea,  ThrottleOpenPctVct);    % Positive since
FlowCms always > 0
        dPresConstrPa    = sign(FlowCms) .* dPresConstrAbsPa;
        Fp  =  p.AreaCrs  .*  (dPressurePa  -  dPresConstrPa);   % Effective
pressure over the grid

        % Gravity forces:
        Fg  =  g  *  p.Fluid.DensityNom  .*  p.LengthPrGrid  .*  p.AreaCrs  .*
sin(p.InclHrzRad); % TODO: Make and Call function that calculates current
Density based on current pressure

        % Fictitious force (since the pipe is the reference frame):
        MassFluid   = p.Volume .* p.Fluid.DensityNom;
        Ffic        = -MassFluid .* AccelerationPipe;    % Fictitious force
has  opposite  sign  as  acceleration  of  reference  frame  (pipe)  when
signconvention is equal

        % Minor forces:     % Future: Add forces resulting from tool joints
and other constrictions in the string/BHA
        Fmin = p.ZeroVectorGridsDs; % = Kmina1 .* (velocity_fluid.^2);



        % Wall shear stress for use in Friction forces:
        if FlowRegime == 'LaminarOnly'

            VelocityPipe = 0;    % Since the pipe motion is the reference
frame
            GeometryPar.RadiusInn = p.RadiusInn;
            GeometryPar.AreaCrs   = p.AreaCrs;
            [wallshearStressPipeLaminar,           Tw_PL_lam]           =
Tw_stringLaminarHBSimple(FlowCms, VelocityPipe, GeometryPar, p.Fluid);
            wallshearStressPipeTotal = wallshearStressPipeLaminar;
            % TA MED DETTE:
            %[TwHbTot,       TwHbTurb,       TwHbLam,       TwPlLam]      =
TwPipeGeoLamTurb(VelLiq,   Re_net,   FlowRegime,   GeometryPar,   FluidPar,
RheologyPar)
            %TwReal = TwHbTot, YpReal, Time, Q)       % For å gjøre den
glatt men likevel med 100% YP-effekt med valgfri verdi ved Q->0.
            %Ff = TwHbTot .* p.AreaSrf;

        elseif FlowRegime == 'LaminarTurb'

            VelocityPipe = 0;    % Since the pipe motion is the reference
frame
            k  = p.Fluid.Viscosity.ConsistencyIndex;
            n  = p.Fluid.Viscosity.FlowBehaviorIndex;
            yp = p.Fluid.Viscosity.YieldPoint;

            % For Turbulent vs Laminar flow: Computing Reynoldsnumber and
weight-number for flow regime:
```

```
            %Ga          = (1./(4*na+2)).^na;        % Parameter valid for
annulus
            G_empty      = (1./(3*n+1)).^n;          % Parameter valid for
empty hole below bit
            %G           = [Ga(1:nBit);  G_empty(nBit+1:nBh)];
            G           = G_empty;

            VelocityLiquid = FlowCms ./ p.AreaCrs - VelocityPipe;
            [ReLam  ReLamNet]  =  fReynoldsNumber(VelocityLiquid,  k,  n,
p.Fluid.DensityNom,  p.RadiusInn,  G);      %  TODO:  p.Fluid.DensityNom  =
p.Fluid.DensityCur
            [FlowRegime,   Re_cr,   Re_ce]   =   fTransition(ReLam,   n,
GlobCstPhysic.ReNmbCstPipe.CritNom,     GlobCstPhysic.ReNmbCstPipe.TurbNom,
GlobCstPhysic.ReNmbCstPipe.Delta_n0, ...

GlobCstPhysic.ReNmbCstPipe.TrnsMinVal);   % TODO: Splitte i to funksjoner:
fReCriticalNCenter og fTransition (Returnerer samme Re_cr og Re_ce hver gang)
            Re_net      = ReLamNet;

            FluidPar.Dns            = p.Fluid.DensityNom;          % TODO:
Use p.Fluid.DensityCur;
            RheologyPar.ModelType   = 'Herschel Bulkley';
            RheologyPar.ViscosityHb.k   = k;
            RheologyPar.ViscosityHb.n   = n;
            RheologyPar.ViscosityHb.yp  = yp;
            RheologyPar.ContinuousSimplified.xi                          =
p.Fluid.RheologyPipe.xi;  % RheologyPipe is comuted for all grids initially
            RheologyPar.ContinuousSimplified.sigma                       =
p.Fluid.RheologyPipe.sigma;
            RheologyPar.ContinuousSimplified.psi                         =
p.Fluid.RheologyPipe.psi;
            RheologyPar.ContinuousSimplified.P_T_default                 =
p.Fluid.RheologyPipe.PT;
            RheologyPar.ContinuousSimplified.omega                       =
p.Fluid.RheologyPipe.Omega;
            RheologyPar.ContinuousSimplified.SmoothFactor                =
p.Fluid.RheologyPipe.SmoothFactor;
            GeometryPar.RaIn = p.RadiusInn;          %GeometryPar.AreaCrs
= p.AreaCrs;

            SimMode = 1;    % Gives absolute YP
            [wallshearStressPipeTotal,   TwTurb,   TwLam,   TwPlLam]   =
TwPipeGeoLamTurb(VelocityLiquid, Re_net, FlowRegime, GeometryPar, FluidPar,
RheologyPar, SimMode);


        else
            'Flow regime type is missing'
            pause
        end



        % Ensuring that the yp-force does not initiate flow when it comes
to rest (due to dicrete time points)
        Ff = p.ZeroVectorGridsDs;
        for i = 1:nGr
            %if abs(FlowLpm(i)) < ZeroFlowLimitLpm    % There is no flow
            %    if abs(FlowLpm(i)) > 0          % Just checking
            %        'stop'
```

```
            %    end

            if abs(FlowLpm(i)) == 0    % OK to test against 0 here because
it i is set to zero

                Fsum = Fp(i) + Fg(i) + Ffic(i) + Fmin(i);

                if abs(wallshearStressPipeTotal(i)) > 0            % Will
never happen
                    'abs(wallshearStressPipeTotal(i)) > 0 - Will never
happen'
                else
                    if yp(i) * p.AreaSrf(i) > abs(Fsum)      % If the yp
has higher potensial than Fsum, it is reduced to Fsum
                        Ff(i) = -Fsum;                        % Ff must be
set directly here to avoid numerical round-off
                        wallshearStressPipeTotal(i) =  Ff(i)/p.AreaSrf(i);
                        %wallshearStressPipeTotal(i) = -Fsum/p.AreaSrf(i);

                    else
                        wallshearStressPipeTotal(i) = -yp(i) * sign(Fsum);
                        Ff(i)     =     wallshearStressPipeTotal(i)     .*
p.AreaSrf(i);
                    end
                end
            else
                Ff(i) = wallshearStressPipeTotal(i) .* p.AreaSrf(i);
            end
        end
        %Ff = wallshearStressPipeTotal .* p.AreaSrf;


        % The derivative dQdt in SI units:
        dDtFlowLpm = CMS2LPM * p.AreaCrs ./ MassFluid .* (Fp + Ff + Fg +
Ffic + Fmin);  % Sign of Ff (Tw) is negative when flow>0




    %   ############   Aligning the outputs    ############

        dxdt = [dDtFlowLpm'; dDtPressureBar'];


    end
```

A.23 PipeFluHrz_2xOrd_Setup

```
    % ------------------------------------------------------------------
-----
    %
    % Setup for Simplified Horizontal Pipe.
    %
    % Maps input units and forms to ODE units and forms
```

```
      % Establish variables for plotting.
      % Calls Init function to data object create structure.
      %
      %  ------------------------------------------------------------------
-----


      function [PipeFluIf, PipeFluObj] = PipeFluHrz_2xOrd_Setup(ObjectName,
InitBoundaries, ParIn, FluidNom, GlobalConstPhysic, GlobalConstSim)

          % Global Constants:
          nGlobalSteps = GlobalConstSim.nGlobalSteps;
          GlobalTimeStep = GlobalConstSim.GlobalTimeStep;

          Solver = 'RungeKutta4';
          %Solver = 'EulerFirstOrder';



          if length(ParIn.LengthPrGrid) > 0
              %'Use a predefined grid-length vector with possible variable
length => Par.nGridsDsFluTotal is ignored'
              % If total length is too long, it will be truncated. If total
length is shorter than needed, a final grid will be added on top with the
length of the residual - NO Wrong.
              nGr                 = ParIn.nGrids;
              p.LengthTotal       = sum(ParIn.LengthPrGrid);
              p.GridLenDsUniform  = [];
              p.LengthPrGrid    = ParIn.LengthPrGrid;

          elseif length(ParIn.LengthPrGrid) == 0  % []
              % Divide the Pipe in a number of grids with equal lengths by
using Par.nGridsDsFluTotal:
              nGr                 = ParIn.nGridsDsFluTotal;
              p.LengthTotal       = PosDsBoundTop - PosDsBoundBot;
              p.GridLenDsUniform  = p.LengthTotal/nGr;
              p.LengthPrGrid    = p.GridLenDsUniform * ones(1, nGr);

          end
          %p.nGridsDs  = nGr;

          p.ZeroVectorGridsDs   = zeros(1, nGr);
          p.UnityVectorGridsDs  = ones(1, nGr);

          % Initial values of State variables set according to Boundaies:
          FlowInitLpm           =           p.UnityVectorGridsDs           *
InitBoundaries.FlowBoundUpStmLpm;
          PresInitBar           =           p.UnityVectorGridsDs           .*
InitBoundaries.PresBoundDnStmBar;
          DensityInit = p.UnityVectorGridsDs .* FluidNom.Density;
          %PresInitBar(nGr) = InitBoundaries.PresBoundTopBar;

          % Flow contriction related
          p.ConstrictionOpenPst   = ParIn.ConstrictionOpenPst;
          p.ThrottleActive        = ParIn.ThrottleActive;
          p.ThrottleGridOrPoint   = ParIn.ThrottleGridOrPoint;

          p.FluidNom = FluidNom;
```

```
          % AVERAGE Drillstring inner diameter of each grid from bottom and
up (Last to first):
          p.RadiusInn     = ParIn.DiameterInnIn./2 * 2.54/100;
          p.InclHrzRad    = (ParIn.InclFromVrtDeg - 90)/180 .* pi();


          % Initiates the Generic Sub-model:
          PipeFluObj  =  PipeFluGen_2xOrd_Init(ObjectName,  p,  FlowInitLpm,
PresInitBar, DensityInit, FluidNom);


      % --- Set the Internal VAriables, Parameters and Constants ---
          %PipeFluObj.Par = p;
          PipeFluObj.Var = [];



          if Solver == 'RungeKutta4'
              PipeFluObj.Cst.Solver   = @Solver_RK4_New;
              PipeFluObj.Cst.nTsLocal                                    =
ceil(GlobalTimeStep/PipeFluObj.Cst.T_max_RK4);      % Future: test on solver
type
              %PipeFluObj.Cst.tLocal                                     =
GlobalTimeStep/PipeFluObj.Cst.nLocalTimeSteps;
              PipeFluObj.Cst.NumStableMode =  1;  % Removes oscillaitons in
RK-steps and delays change of sign of states
              PipeFluObj.Cst.NumericStabilizationMode = 1; %optional input
parameter that may be set to one in order to remove ocsillations and change
of sign between the RK iterations

          elseif Solver == 'EulerFirstOrder'
              PipeFluObj.Cst.Solver   = @Euler1;
              PipeFluObj.Cst.nTsLocal                                    =
ceil(GlobalTimeStep/PipeFluObj.Cst.T_max_Euler);
              %PipeFluObj.Cst.tLocal                                     =
GlobalTimeStep/PipeFluObj.Cst.nTsLocal;
              PipeFluObj.Cst.NumStableMode =  1;  % Removes oscillaitons in
RK-steps and delays change of sign of states
              PipeFluObj.Cst.NumericStabilizationMode = 0;
          end
          PipeFluObj.Cst.TsLocal                                         =
GlobalTimeStep/PipeFluObj.Cst.nTsLocal;



          %ZeroFlowLimitLpm = GlobCstPhysic.ZeroFlowLimitLpm;
          MassFluid                   =       PipeFluObj.Par.Volume       .*
PipeFluObj.Par.Fluid.DensityNom;
          if 0
              yp = PipeFluObj.Par.Fluid.Viscosity.YieldPoint;
              ZeroFlowLimitCms        =        max(GlobalTimeStep         .*
PipeFluObj.Par.AreaCrs ./MassFluid .* yp .* PipeFluObj.Par.AreaSrf);
          else
```

```
            dPresPrTimeStepPa    =    PipeFluObj.Par.Fluid.BulkModulus    ./
PipeFluObj.Par.Volume     .*     GlobalConstPhysic.dFlowInMaxPrSecCms     .*
GlobalTimeStep;
            dForcePrTimeStep             =       dPresPrTimeStepPa       .*
PipeFluObj.Par.AreaCrs;
            ZeroFlowLimitCms         =         max(GlobalTimeStep         .*
PipeFluObj.Par.AreaCrs ./MassFluid .* dForcePrTimeStep) / 1;
        end
        ZeroFlowLimitLpm = ZeroFlowLimitCms * 60000 / 10000;
        disp(['ZeroFlowLimitLpm = ', num2str(ZeroFlowLimitLpm)]);
        PipeFluObj.Par.ZeroFlowLimitLpm = ZeroFlowLimitLpm;



        % Set the default Initial Inputs (needed for setting InputsPrev in
first time step):
        PipeFluIf.Inputs.FlowUpStmLpm                                    =
InitBoundaries.FlowBoundUpStmLpm;
        PipeFluIf.Inputs.PresDnStmBar                                    =
InitBoundaries.PresBoundDnStmBar;   % Initialize with a boundary that gives
steady conditions
        PipeFluIf.Inputs.ThrottleClosePct = 0;
        PipeFluIf.Inputs.VelDsAxl = 0;
        PipeFluIf.Inputs.VelDsAng = 0;
        PipeFluIf.Inputs.AccDsAxl = 0;
        PipeFluIf.Inputs.AccDsAng = 0;



        % States and additional variables in plot-order:
        OutputsInitial             =             [PipeFluObj.States.Flow;
PipeFluIf.Inputs.FlowUpStmLpm;                PipeFluIf.Inputs.PresDnStmBar;
PipeFluObj.States.Pres; DensityInit'];
        pIntFc.OutputOrder   = length(OutputsInitial);
        PipeFluIf.PlotMatrix = [OutputsInitial, zeros(pIntFc.OutputOrder,
nGlobalSteps-1)];



        % --- Setup for plotting ---
        %pIntFc.PlotGrouping = [nGr+1, nGr+1, nGr];        % nGr+1 for P
and Q?
        %pIntFc.OutputNames    = {'Flow  rates  (LPM)','Pressure  (Bar)',
'Density'};   % , 'Wall shear stress', 'Frictional pressure drop'
        pIntFc.PlotGrouping = [nGr+1, nGr+1];       % nGr+1 for P and Q?
        pIntFc.OutputNames  = {'Flow rates (LPM)','Pressure (Bar)'};


        % --- Set the Outputs ---

        PipeFluIf.Par = pIntFc;

        PipeFluIf.Outputs.Flow               =    [PipeFluObj.States.Flow;
PipeFluIf.Inputs.FlowUpStmLpm];
        PipeFluIf.Outputs.Pres         =   [PipeFluIf.Inputs.PresDnStmBar;
PipeFluObj.States.Pres];
        PipeFluIf.Outputs.Dens    = p.ZeroVectorGridsDs';
        %PipeFluIf.Outputs.FlowLeak    = p.ZeroVectorGridsDs';
```

```
        %PipeFluIf.Outputs.ShearStressAxl = ShearStressAxl;
        %PipeFluIf.Outputs.PresFrictionComp = PresFrictionComp;


    end
```

A.24 PipeFluHrz_InputSignalGenerator.m

```
    %-------------------------------------------------------------------
------
    % Generates smooth input signals from step/hold values
    % Smoothening can be done by a 1. order or 2. order filter
    %-------------------------------------------------------------------
------

    function [FlowrateInput_1Order_0_2000_LPM, ThrottleInput_ClosePct] =
PipeFluHrz_InputSignalGenerator(GlobalConstSim, GlobConstPhys)


        % Generate Flowrate Input (from Main pump)
        AmplNom         = 2000;
        startValue      = 0;
        holdTimes       = [1; 20];
        holdValues      = [AmplNom; AmplNom];
        filterOrder     = 2;
        filterTime        = 1;    % Suitable for 2. order dynamics (w0 =
1/filterTime)
        [FlowrateInput_1Order_0_2000_LPM,
FlowrateInput_1Order_0_2000_Integral]    =    fRampAndHold3(GlobalConstSim,
filterOrder, filterTime, startValue, holdTimes, holdValues);
        InputSignals.FlowrateIn = FlowrateInput_1Order_0_2000_LPM;
        dFlowInMaxPrSecLpm = AmplNom/filterTime;
        GlobConstPhys.dFlowInMaxPrSecCms = dFlowInMaxPrSecLpm / 60000;



        % General Throttle/Constriction Input signal:
        startValue  = 0;
        holdTimes   = [45, 2];
        holdValues  = [95, 0];  % Lavere enn 4 gir ustblit system ved bruk
på DsMain (uavh av dT)
        filterOrder = 1;
        filterTime  = 0.25;
        [ThrottleInput_ClosePct,    ThrottlingInput_OpenPctIntegral]    =
fRampAndHold3(GlobalConstSim,    filterOrder,    filterTime,    startValue,
holdTimes, holdValues);
        InputSignals.ThrottleValveClosePct = ThrottleInput_ClosePct;



        % Flowrate Output (for Tank, Lift pump etc)
        %startValue      = 0;
        %holdTimes       = [20; 20];
        %holdValues      = [4000; 0];
        %filterOrder     = 2;
```

```
          %filterTime      = 2.5;   % 10 is Suitable for 1. ord. dyn; 2.5 is
Suitable for 2. ord dyn. (w0 = 1/filterTime)

          %[FlowOut_Dynamic_Lpm, FlowOut_Dynamic_Lpm_Integral] ...
          %       = fRampAndHold3(GlobalConstSim, filterOrder, filterTime,
startValue, holdTimes, holdValues);


      end
```

A.25 PipeFluHrz_SemiImplicitPde_Setup.m

```
      % ------------------------------------------------------------------
-----
      % Setup for Simplified Horizontal Pipe.
      %
      % Maps input units and forms to ODE units and forms
      % Establish variables for plotting.
      % Calls Init function to data object create structure.
      % ------------------------------------------------------------------
-----

      function             [PipeFluIf,             PipeFluObj]             =
PipeFluHrz_SemiImplicitPde_Setup(ObjectName,     InitBoundaries,     ParIn,
FluidNom, GlobalConstPhysic, GlobalConstSim)
              % ExternalIfc, InternalObj

          % Global Constants:
          nGlobalSteps = GlobalConstSim.nGlobalSteps;
          GlobalTimeStep = GlobalConstSim.GlobalTimeStep;


          if length(ParIn.LengthPrGrid) > 0
              %'Use a predefined grid-length vector with possible variable
length => Par.nGridsDsFluTotal is ignored'
              nGr                 = ParIn.nGrids;
              p.LengthTotal       = sum(ParIn.LengthPrGrid);
              p.GridLenDsUniform  = [];
              p.LengthPrGrid    = ParIn.LengthPrGrid;
          elseif length(ParIn.LengthPrGrid) == 0
              '';
          end


          p.ZeroVectorGridsDs   = zeros(1, nGr);
          p.UnityVectorGridsDs  = ones(1, nGr);

          % Initial values of State variables set according to Boundaies -
Movw to INIT?
          FlowInitLpm           =           p.UnityVectorGridsDs           *
InitBoundaries.FlowBoundUpStmLpm;
          PresInitBar           =          p.UnityVectorGridsDs          .*
InitBoundaries.PresBoundDnStmBar;
          DensityInit = p.UnityVectorGridsDs .* FluidNom.Density;
```

```
            % Flow contriction related
            p.ConstrictionOpenPst   = ParIn.ConstrictionOpenPst;
            p.ThrottleActive        = ParIn.ThrottleActive;
            p.ThrottleGridOrPoint = ParIn.ThrottleGridOrPoint;

            p.FluidNom = FluidNom;

            % AVERAGE Drillstring inner diameter of each grid from bottom and
up (Last to first):
            p.RadiusInn     = ParIn.DiameterInnIn./2 * 2.54/100;
            p.InclHrzRad    = (ParIn.InclFromVrtDeg - 90)/180 .* pi();


             % Initiates variables i the Generic Pde based sub-model:
            PipeFluObj   =   PipeFlu_PdeGen_Init(ObjectName,   p,   FlowInitLpm,
PresInitBar, DensityInit, FluidNom);


            % --- Set the Internal VAriables, Parameters and Constants ---
            %PipeFluObj.Par = p;
            PipeFluObj.Var = [];


            % Set Solver etc
            PipeFluObj.Cst.Solver   = @Solver_PdeSemiImplicit;
            PipeFluObj.Cst.nTsLocal     =     ceil(GlobalTimeStep/1);         %
PipeFluObj.Cst.T_max_RK4
            %PipeFluObj.Cst.tLocal                                             =
GlobalTimeStep/PipeFluObj.Cst.nLocalTimeSteps;
            PipeFluObj.Cst.TsLocal                                            =
GlobalTimeStep/PipeFluObj.Cst.nTsLocal;


            %ZeroFlowLimitLpm = GlobCstPhysic.ZeroFlowLimitLpm;
            MassFluid                       =       PipeFluObj.Par.Volume       .*
PipeFluObj.Par.Fluid.DensityNom;
            if 0
                yp = PipeFluObj.Par.Fluid.Viscosity.YieldPoint;
                ZeroFlowLimitCms        =        max(GlobalTimeStep         .*
PipeFluObj.Par.AreaCrs ./MassFluid .* yp .* PipeFluObj.Par.AreaSrf);
            else
                dPresPrTimeStepPa    =    PipeFluObj.Par.Fluid.BulkModulus    ./
PipeFluObj.Par.Volume     .*     GlobalConstPhysic.dFlowInMaxPrSecCms     .*
GlobalTimeStep;
                dForcePrTimeStep             =       dPresPrTimeStepPa       .*
PipeFluObj.Par.AreaCrs;
                ZeroFlowLimitCms        =        max(GlobalTimeStep         .*
PipeFluObj.Par.AreaCrs ./MassFluid .* dForcePrTimeStep) / 1;
            end
            ZeroFlowLimitLpm = ZeroFlowLimitCms * 60000 / 10000;
            disp(['ZeroFlowLimitLpm = ', num2str(ZeroFlowLimitLpm)]);
            PipeFluObj.Par.ZeroFlowLimitLpm = ZeroFlowLimitLpm;



            % Set the default Initial Inputs (needed for setting InputsPrev in
first time step):
```

```
          PipeFluIf.Inputs.FlowUpStmLpm                                    =
InitBoundaries.FlowBoundUpStmLpm;
          PipeFluIf.Inputs.PresDnStmBar                                    =
InitBoundaries.PresBoundDnStmBar;   % Initialize with a boundary that gives
steady conditions
          PipeFluIf.Inputs.ThrottleClosePct = 0;
          PipeFluIf.Inputs.VelDsAxl = 0;
          PipeFluIf.Inputs.VelDsAng = 0;
          PipeFluIf.Inputs.AccDsAxl = 0;
          PipeFluIf.Inputs.AccDsAng = 0;



          % States and additional variables in plot-order:
          OutputsInitial              =             [PipeFluObj.States.Flow;
PipeFluIf.Inputs.FlowUpStmLpm;                PipeFluIf.Inputs.PresDnStmBar;
PipeFluObj.States.Pres; DensityInit'];
          pIntFc.OutputOrder   = length(OutputsInitial);
          PipeFluIf.PlotMatrix = [OutputsInitial, zeros(pIntFc.OutputOrder,
nGlobalSteps-1)];



          % --- Setup for plotting ---
          %pIntFc.PlotGrouping = [nGr+1, nGr+1, nGr];        % nGr+1 for P
and Q?
          %pIntFc.OutputNames    = {'Flow rates (LPM)','Pressure (Bar)',
'Density'};   % , 'Wall shear stress', 'Frictional pressure drop'
          pIntFc.PlotGrouping = [nGr+1, nGr+1];       % nGr+1 for P and Q?
          pIntFc.OutputNames  = {'Flow rates (LPM)','Pressure (Bar)'};


          % --- Set the Outputs ---
          PipeFluIf.Par = pIntFc;

          PipeFluIf.Outputs.Flow               =    [PipeFluObj.States.Flow;
PipeFluIf.Inputs.FlowUpStmLpm];
          PipeFluIf.Outputs.Pres         =  [PipeFluIf.Inputs.PresDnStmBar;
PipeFluObj.States.Pres];
          PipeFluIf.Outputs.Dens    = p.ZeroVectorGridsDs';

      end
```

A 26 PipeFluHrz_Step.m

```
      % ------------------------------------------------------------------
-----
      % This function returns Output values for the next time step of a
dynamic
      % model. It should be called directly from a Master algorithm that
controls
      % the execution time (main time-loop).
      %
```

```
      % The model may consist of Sub-models, and the outputs may be States
from
      % ODE's or additional simulation variables computed from static
equations
      % (typically with States as inputs).
      %
      % This Step function combines States and parameters of the Sub-models
into
      % combined vectors and structures before calling a Solver with the
Combined
      % ODE model as argument. I.e., the entire combined model is solved
      % simultaneously by the solver.
      %
      % The returning State values for the combined model are then splitted
      % into states for the Sub-models. And if additional simulation variables
      % are needed by the Master/other Sub-models, they are computed. Both
States
      % and additional simulation variables are returnd in a separate Output
      % structure, while the Internal States are stored and updated in a
      % Structure whose internal structure is unknown for the Master.
      %
      % The Master also has to set the correct Inputs, including Inputs from
      % the previous time step. The latter is needed in order to compute
      % time-derivatives (for increased stability). This is done in a generic
way
      % for all inputs.
      %
      % (The Master is responsible for pairing inputs and outputs
      % between all Sub-models based on confiruration of the total system.)
      %
      % NOTE for making scenarios by DSB:
      % When DSB modifies Parameters, the original values should be stored
in a
      % copy structure and put back when scenario is finished.
      % The model will then recover naturally. Consider also storing States
and
      % put these back for some special slow operations (instructor may
decide)
      %
      % DelayedChangeOfSign is..
      % TODO: Include Turbulence
      % ------------------------------------------------------------------
-----

      function [Outputs, PipeFluObj] = PipeFluHrz_Step(Inputs, InputsPrev,
PipeFluObj, GlobalConstPhysic, GlobalConstSim, SolverType)


          GlobalTimeStep = GlobalConstSim.GlobalTimeStep;
          TsLocal     = PipeFluObj.Cst.TsLocal;
          nTsLocal    = PipeFluObj.Cst.nTsLocal;

          Solver      = PipeFluObj.Cst.Solver;


          EventBuilder = 0;
          if EventBuilder == 1        % Flow restriciton (Pack-off etc)
              PipeFluObj.Par.AreaCrs    =    PipeFluObj.Par.AreaCrsOrg    -
(PipeFluObj.Par.AreaCrsOrg     .*     PipeFluObj.Par.ThrottleActive     .*
Inputs.ThrottleClosePct/100);
```

```
            PipeFluObj.Par.RadiusInn = sqrt(PipeFluObj.Par.AreaCrs/pi);
            PipeFluObj.Par.AreaSrf   = pi * 2*PipeFluObj.Par.RadiusInn .*
PipeFluObj.Par.LengthPrGrid;
            PipeFluObj.Par.Volume          =   PipeFluObj.Par.AreaCrs   .*
PipeFluObj.Par.LengthPrGrid;
        end


        % ---    Preparing inputs - Note they are for total object (no
splitting needed here)   ---

        % Organize scalar inputs into vectors and find derivatives:
        InputsScalarsCurr    = [Inputs.FlowUpStmLpm; Inputs.PresDnStmBar;
Inputs.ThrottleClosePct];
        InputsScalarsPrev                 =     [InputsPrev.FlowUpStmLpm;
InputsPrev.PresDnStmBar; InputsPrev.ThrottleClosePct];
        InputsScalarsDerv    = (InputsScalarsCurr - InputsScalarsPrev) ./
GlobalTimeStep;  % Computing derivatives

        %  Align  VECTOR  inputs  to  internal  grid  structure  and  find
derivatives:
        nGrTot  = PipeFluObj.Par.nGridsDs;  % = DsMainFluid.Par.nGridsDs +
1;
        VelDsAxlCurr                                                    =
AlignVectorsValuesToMultipleLength(Inputs.VelDsAxl, nGrTot);
        VelDsAxlPrev                                                    =
AlignVectorsValuesToMultipleLength(InputsPrev.VelDsAxl, nGrTot);
        VelDsAngCurr                                                    =
AlignVectorsValuesToMultipleLength(Inputs.VelDsAng, nGrTot);
        VelDsAngPrev                                                    =
AlignVectorsValuesToMultipleLength(InputsPrev.VelDsAng, nGrTot);

        AccDsAxlCurr                                                    =
AlignVectorsValuesToMultipleLength(Inputs.AccDsAxl, nGrTot);
        AccDsAxlPrev                                                    =
AlignVectorsValuesToMultipleLength(InputsPrev.AccDsAxl, nGrTot);
        AccDsAngCurr                                                    =
AlignVectorsValuesToMultipleLength(Inputs.AccDsAng, nGrTot);
        AccDsAngPrev                                                    =
AlignVectorsValuesToMultipleLength(InputsPrev.AccDsAng, nGrTot);

        % Assembling all input vectors in matrixes:
        InputsPrGridCurr     = [VelDsAxlCurr, VelDsAngCurr, AccDsAxlCurr,
AccDsAngCurr];
        InputsPrGridPrev     = [VelDsAxlPrev, VelDsAngPrev, AccDsAxlPrev,
AccDsAngPrev];
        InputsPrGridDerv      = (InputsPrGridCurr - InputsPrGridPrev) ./
GlobalTimeStep;  % Computing derivatives

        % Find Accelration of Ds based on derivative computed here and add
to matrix of vectors:
        AccDsAxlCalc = InputsPrGridDerv(:,1);
        %InputsPrGridCurr = [InputsPrGridCurr, AccDsAxlCalc];
        InputsPrGridPrev = [InputsPrGridPrev, AccDsAxlCalc];
        InputsPrGridDerv               =               [InputsPrGridDerv,
PipeFluObj.Par.ZeroVectorGridsDs'];

        % Find first local inputs:
        InputsScalarsLocal  = InputsScalarsPrev;     % Start with previous
(Local increments are added before calling the Solver in the local time loop)
```

```
            InputsPrGridLocal   = InputsPrGridPrev;      % Start with previous
(Local increments are added before calling the Solver in the local time loop)


            % ---   Calling the solver and Step-function   ---

            % ODE models
            if strcmp(SolverType, "ODE")
                NumStableMode = PipeFluObj.Cst.NumStableMode;
                DelayedChangeOfSign = 0;    % Value 1 ensures that all variables
go to zero before changing sign (reduces ocsillations for fluids with a yield
point)

                BiasFlowDs = PipeFluObj.Par.BiasFlowDs;
                BiasPresDs = PipeFluObj.Par.BiasPresDs;
                BiasDensDs = PipeFluObj.Par.BiasDensDs;

                % Organizing States into one vector, and parameters of Sub-
models into one common Struct
                StateVecLocal           =           [PipeFluObj.States.Flow;
PipeFluObj.States.Pres];

                % Keep the current flow for checking ChangeOfSign after Step()
                FlowCurrent                                                 =
StateVecLocal(BiasFlowDs+1:BiasFlowDs+PipeFluObj.Par.nGridsDs);

                for i = 1:nTsLocal
                    InputsScalarsLocal      =      InputsScalarsLocal      +
InputsScalarsDerv * TsLocal;
                    InputsPrGridLocal   = InputsPrGridLocal + InputsPrGridDerv
* TsLocal;
                    StateVecLocal       = Solver(@DsMain_Horizontal_2xOrd_ODE,
InputsScalarsLocal,   InputsPrGridLocal,   StateVecLocal,   PipeFluObj.Var,
PipeFluObj.Par, GlobalConstPhysic, TsLocal, NumStableMode);



                    % Error checking
                    if      NanAndInfCheck(StateVecLocal,      'StateVecLocal
DsFlu_Combined_ODE')
                        'NanAndInfCheck = True';
                    end
                    % Error checking
                    if DelayedChangeOfSign == 1
                        FlowNew                                             =
StateVecLocal(BiasFlowDs+1:BiasFlowDs+PipeFluObj.Par.nGridsDs);
                        ChangeOfSign = sign(FlowCurrent) .* sign(FlowNew);
                        for j=1:PipeFluObj.Par.nGridsDs
                            if ChangeOfSign(j) < 0
                                disp('Change    Of    Sign    is    delayed');
%nDelayedChangeOfSign = nDelayedChangeOfSign + 1;
                            end
                        end
                        FlowToZeroOrNot = ChangeOfSign >= 0;
                        FlowNew = FlowNew .* FlowToZeroOrNot;

StateVecLocal(BiasFlowDs+1:BiasFlowDs+PipeFluObj.Par.nGridsDs) = FlowNew;
                        FlowCurrent = FlowNew;
                    end
                end
```

```
            % Error checking
            if abs(InputsScalarsLocal - InputsScalarsCurr) > 1E-6
                disp('Error in interpolating inputs')
            end

            %States are updated in the sub-object structures
            PipeFluObj.States.Flow                                          =
StateVecLocal(BiasFlowDs+1:BiasFlowDs+PipeFluObj.Par.nGridsDs);
            PipeFluObj.States.Pres                                          =
StateVecLocal(BiasPresDs+1:BiasPresDs+PipeFluObj.Par.nGridsDs);


        % PDE models
        elseif strcmp(SolverType, "PDE")
            for i = 1:nTsLocal
                VelLiq = PipeFluObj.States.Flow ./ PipeFluObj.Par.AreaCrs;
                VelGas = PipeFluObj.States.Flow * 0;
                Pres = PipeFluObj.States.Pres;
                InputsScalarsLocal      =      InputsScalarsLocal      +
InputsScalarsDerv * TsLocal;
                InputsPrGridLocal  = InputsPrGridLocal + InputsPrGridDerv
* TsLocal;
                [Pres,            VelLiq,            VelGas]            =
PipeFlu_SemiImplicitPDE(InputsScalarsLocal,    InputsPrGridLocal,    VelLiq,
VelGas, Pres, ...

PipeFluObj.Var, PipeFluObj.Par, GlobalConstPhysic, TsLocal);
                %StateVecLocal      = Solver(@DsMain_Horizontal_2xOrd_ODE,
InputsScalarsLocal,   InputsPrGridLocal,   StateVecLocal,   DsFluTotal.Var,
DsFluTotal.Par, GlobalConstPhysic, TsLocal, NumStableMode);
            end

        else
            '';
        end


        VelocityPipe = 0;   % Since the pipe motion is the reference frame
        PipeFluObj.Var.DensDyn                                              =
CalcFluidDensityFromEqOfState(PipeFluObj.States.Pres*GlobalConstPhysic.Bar2
Pa,   PipeFluObj.Par.FluidNom.PresRefPa,   PipeFluObj.Par.FluidNom.Density,
PipeFluObj.Par.FluidNom.BulkModulus);
        ThrottleOpenPctVct  =  100 * PipeFluObj.Par.UnityVectorGridsDs -
(PipeFluObj.Par.ThrottleActive .* Inputs.ThrottleClosePct);


        % Setting Outputs for Total model - Only these should be available
for Master:
        Outputs.Flow      = [PipeFluObj.States.Flow; Inputs.FlowUpStmLpm];
        Outputs.Pres      = [Inputs.PresDnStmBar; PipeFluObj.States.Pres];
        Outputs.Dens        = PipeFluObj.Var.DensDyn;
        Outputs.ThrottleOpenPctVct = ThrottleOpenPctVct;

    end


        % Determine internal time step (flytt fra StepGeneric) and
interpolate inputs:
```

```
            % TODO: Copy from Old Step function but utilize derivatives (if
non-zero)


          % Call the Generic Step function with a Solver, Internal time step
and ODE-Model as input:
            % Repeate until a global time step is simulated
            % Note: In case of several sub-models, it is the total model
that
            % is called here (decpoupling and coupling of variables are
done in the Total ODE-Model)
```

A.27 Tw_stringNewtonianLaminar.m

```
      %
**********************************************************************
      %
      % The function returns average wall shear stress values (Tw) for
      % Newtonian fluids in circular PIPES in LAMINAR flow
      % when the bulk flow rate (v) is the dynamic input.
      %
      % Vectors where the elements represent each pipe segment/grid can be
      % provided as input => the output will be a vector.
      %
      % Future extensions:
      % Include laminar flow for Bingham plasic (BP) fluids and power law
(PL) fluids.
      %
      % Note:
      % The frictional pressure gradient (Pa/m) is: dp/dz = 2/RaInn * Tw
      % - This can be derived from Eq. 3.2.8b in Gjerstad 2014 PhD thesis -
Simplified Flow Equations for Single-Phase non-Newtonian Fluids in Couette-
Poiseuille Flow and in Pipes
      %
      %
**********************************************************************

      %
      function  [Tw]  =  Tw_stringNewtonianLaminar(Flowrate,  velocityPipe,
GeometryPar, FluidParameters)

      % If inputs are vectors, each parameter within the Structure must be a
vector (not a vector of Structures)


          RaInn   = GeometryPar.RadiusInn;
          AreaCrs = GeometryPar.AreaCrs;
          k       = FluidParameters.Viscosity.ConsistencyIndex;
          %n       = FluidParameters.Viscosity.FlowBehaviorIndex;
          %yp      = FluidParameters.Viscosity.YieldPoint;


          %Flow inside string relative to string velocity (m/s):
          velocityFluid = Flowrate./AreaCrs - velocityPipe;
```

```
            % Wall shear stress for Newtonian fluid in laminar flow:
            %
            Tw = -velocityFluid;    % STUDENT: THIS IS JUST A DUMMY FUNCTION -
> REPLACE THIS BY THE CORRECT FORMULA - USE HAGEN-POISEUILLE EQ +
            %   REF Eqs. 3.2.5 - 3.2.8 in Gjerstad 2014 PhD thesis - Simplified
Flow Equations for Single-Phase non-Newtonian Fluids in Couette-Poiseuille
Flow and in Pipes
            %   AND REPLACE THE AREAS BY RADIUS ETC.

        end
```

Appendix B Kjell Kåre Fjelde Model

B.1 main17042023

```
% Transient two-phase code based on AUSMV scheme: Gas and Water
% The code assumes uniform geometry

% time - Seconds

% p - pressure at new time level (Pa)
% dl - density of liquid at new time level (kg/m3)
% dg - density of gas at new time level (kg/m3)
% eg - phase volume fraction of gas at new time level (0-1)
% ev - phase volume fraction of liquid at new time level (0-1)
% vg - phase velocity of gas at new time level (m/s)
% vl - phase velocity of liquid at new time level (m/s)
% qv - conservative variables at new time level  ( 3 in each cell)
% temp - temperature in well (K)

% po - pressure at old time level (Pa)
% dlo - density of liquid at old time level (kg/m3)
% dgo - density of gas at new old level (kg/m3)
% ego - phase volume fraction of gas at old time level (0-1)
% evo - phase volume fraction of liquid at old time level (0-1)
% vgo - phase velocity of gas at old time level (m/s)
% vlo - phase velocity of liquid at old time level (m/s)
% qvo - conservative variables at old  time level  ( 3 in each cell)
% temp - temperature in well (K)




clear;
t = cputime
tic,

% Geometry data/ Must be specified
welldepth = 4000;
nobox = 25; %Number of boxes in the well

% Note that one can use more refined grid, 50, 100 boxes.
% When doing this, remember to reduce time step to keep the CFL number
% fixed below 0.25.. dt < cfl x dx/ speed of sound in water. If boxes
are
% doubled, then half the time step.

nofluxes = nobox+1;  % Number of cell boundaries
dx = welldepth/nobox; % Boxlength
%dt = 0.005;

% Welldepth. Cell 1 start at bottom
x(1)= -1.0*welldepth+0.5*dx;
for i=1:nobox-1
 x(i+1)=x(i)+ dx;
end
```

```
      % VERY IMPORTANT: BELOW THE TIMESTEP IS SET. MAKE SURE THAT THE
      % CFL CONDIDTION IS FULFILLED. IF NUMBER OF BOXES IS CHANGED. DX WILL
      % CHANGE AND DT HAS TO BE ADJUSTED TO KEEP THE CFL NUMBER FIXED.

      dt= 0.02;  % Timestep (seconds)
      dtdx = dt/dx;
      time = 0.0; % initial time.
      endtime = 300; % Time for ending simulation  (seconds)
      nosteps = endtime/dt;  %Number of total timesteps. Used in for loop.
      timebetweensavingtimedata = 0.1;  % How often in s we save data vs time
for plotting.
      nostepsbeforesavingtimedata = timebetweensavingtimedata/dt;

      % Slip parameters used in the gas slip relation. vg =Kvmix+S
      k = 1.2;
      s = 0.55;

      % Boundary condition at outlet
      pbondout=1000000; % Pascal  (1 bar)


      % Initial temperature distribution. (Kelvin)
      % Note that this is only used if we use density models that depend on
      % temperature

      tempbot = 110+273;
      temptop = 50+273;
      tempgrad= (tempbot-temptop)/welldepth;
      tempo(1)=tempbot-dx/2*tempgrad;
      for i = 1:nobox-2
        tempo(i+1)=tempo(i)-dx*tempgrad;
      end
      tempo(nobox)=tempo(nobox-1)-dx*tempgrad;

      temp = tempo;

      % Different fluid density parameters
      % Note how we switch between different models later.
      % These parameters are used when finding the
      % primitive variables pressure, densities in an analytical manner.
      % Changing parameters here, you must also change parameters inside the
      % density routines roliq and rogas.

      % Simple Water density model & Ideal Gas. See worknote Extension of
AUSMV
      % scheme.

      rho0=1000;  % Water density at STC (Standard Condition) kg/m3
      Bheta=2.2*10^9; % Parameter that depend on the compressibility of water
      Alpha=0.000207; % Parameter related to thermal expansion/compression
      R = 286.9; % Ideal gas parameter
      P0=101000; % Pressure at STC (Pa)
      T0=15+273.15; % Temperature at STC (K)


      % Very simple models (PET510 compendium)
```

```
      al = 1500; % Speed of sound in water.
      rt= 100000; % Ideal gas parameter in model rhog = p/rt  (rt = ag^2)
      rho0=1000; % Water density at STC (Standard Condition) kg/m3
      P0=101000; % Pressure at STC (Pa)
      T0=15+273.15; % Temperature at STC (K)


      % Viscosities (Pa*s)/Used in the frictional pressure loss model
(dpfric).
      viscl = 0.5; % Liquid phase
      viscg = 0.0000182; % Gas phase




      % Gravity constant

        g = 9.81; % Gravitational constant m/s2

        g = 0;

      % Well opening. opening = 1, fully open well, opening = 0 (<0.01), the
well
      % is fully closed. This variable will control what boundary conditions
that
      % will apply at the outlet (both physical and numerical): We must change
      % this further below in the code if we want to change status on this.

        wellopening = 1.0;  % This variable determines if
      %the well is closed or not, wellopening = 1.0 -> open. welllopening =
0
      %-> Well is closed. This variable affects the boundary treatment.

        bullheading = 0.0; % This variable can be set to 1.0 if we want to
simulate
      % a bullheading operation. But the normal is to set this to zero.


      % Specify if the primitive variables shall be found either by
      % a numerical or analytical approach. If analytical = 1, analytical
      % solution is used. If analytical = 0. The numerical approach is used.
      % using the itsolver subroutine where the bisection numerical method
      % is used. We use analytical.

        analytical = 1;




      % Initialization of rest of geometry.
      % Here we specify the outer and inner diameter and the flow area
      % We assume 12.25 x 5 inch annulus. But this can be modified.
```

```
   for i = 1:nobox

    do(i)=0.331;
    di(i) = 0.127;

    area(i) = 3.14/4*(do(i)*do(i)- di(i)*di(i));

   end
```

```
% Initialization of slope limiters. These are used for
% reducing numerical diffusion and will be calculated for each timestep.
% They make the numerical scheme second order.
  for i = 1:nobox
    sl1(i)=0;
    sl2(i)=0;
    sl3(i)=0;
    sl4(i)=0;
    sl5(i)=0;
    sl6(i)=0;
  end
```

```
% Now comes the intialization of the physical variables in the well.
% First primitive variables, then the conservative ones.
```

```
% Below we intialize pressure and fluid densities. We start from top
of
% the well and calculated downwards. The calculation is done twice with
% updated values to get better approximation. Only hydrostatic
% considerations since we start with a static well.

for i = 1:nobox
  eg(i)=0.0;  % Gas volume fraction
  ev(i)=1-eg(i); % Liquid volume fraction
end

p(nobox)= pbondout+0.5*g*dx*...
    (ev(nobox)*rholiq(P0,T0)+eg(nobox)*rogas(P0,T0));      %  Pressure
(Pa)
dl(nobox)=rholiq(p(nobox),tempo(nobox));  % Liquid density kg/m3
dg(nobox)=rogas(p(nobox),tempo(nobox));   % Gas density kg/m3
```

```
for i=nobox-1:-1:1
p(i)=p(i+1)+dx*g*(ev(i+1)*dl(i+1)+eg(i+1)*dg(i+1));
```

```
      dl(i)=rholiq(p(i),tempo(i));
      dg(i)=rogas(p(i),tempo(i));
      end

       for i=nobox-1:-1:1
        rhoavg1= (ev(i+1)*dl(i+1)+eg(i+1)*dg(i+1));
        rhoavg2= (ev(i)*dl(i)+eg(i)*dg(i));
        p(i)=p(i+1)+dx*g*(rhoavg1+rhoavg2)*0.5;
        dl(i)=rholiq(p(i),tempo(i));
        dg(i)=rogas(p(i),tempo(i));

       end

      % Intitialize phase velocities, volume fractions, conservative
variables
      % and friction and hydrostatic gradients.
      % The basic assumption is static fluid, one phase liquid.

      for i = 1:nobox
        vl(i)=0; % Liquid velocity new time level.
        vg(i)=0; % Gas velocity at new time level
        eg(i)=0.0;  % Gas volume fraction
        ev(i)=1-eg(i); % Liquid volume fraction
        qv(i,1)=dl(i)*ev(i)*area(i);  % Conservative variable for liquid mass
(kg/m)
        qv(i,2)=dg(i)*eg(i)*area(i);   % Conservative variable for gas mass
(kg/m)
        qv(i,3)=(dl(i)*ev(i)*vl(i)+dg(i)*eg(i)*vg(i))*area(i);             %
Conservative variable for mixture moementum
        fricgrad(i)=0;   % Pa/m
        hydgrad(i)=g*(dl(i)*ev(i)+eg(i)*dg(i));  % Pa/m
      end


      % Section where we also initialize values at old time level


      for i=1:nobox
        dlo(i)=dl(i);
        dgo(i)=dg(i);
        po(i)=p(i);
        ego(i)=eg(i);
        evo(i)=ev(i);
        vlo(i)=vl(i);
        vgo(i)=vg(i);
        qvo(i,1)=qv(i,1);
        qvo(i,2)=qv(i,2);
        qvo(i,3)=qv(i,3);
      end


      % Intialize fluxes between the cells/boxes

      for i = 1:nofluxes
        for j =1:3
         flc(i,j)=0.0; % Flux of liquid over box boundary
         fgc(i,j)=0.0; % Flux of gas over box boundary
         fp(i,j)= 0.0; % Pressure flux over box boundary
        end
```

```
end


% Main program. Here we will progress in time. First som intializations
% and definitions to take out results. The for loop below runs until
the
% simulation is finished.

countsteps = 0;
counter=0;
printcounter = 1;
pin(printcounter) = (p(1)+dx*0.5*hydgrad(1))/100000; % Pressure in bar
at bottom for time storage
pout(printcounter)= pbondout/100000; % Pressure at outlet of uppermost
cell
pnobox(printcounter)= p(nobox)/100000; % Pressure in middle of
uppermost cell
liquidmassrateout(printcounter) = 0; % liquid mass rate at outlet kg/s
gasmassrateout(printcounter)=0; % gass mass rate at outlet kg/s
timeplot(printcounter)=time; % Array for time and plotting of
variables vs time
pitvolume=0;
pitrate =0;
pitgain(printcounter)=0;

kickvolume=0;
bullvolume=0;


% The temperature is not updated but kept fixed according to the
% initialization.


% Now comes the for loop that runs forward in time. This is repeated
for
% every timestep.

for i = 1:nosteps
   countsteps=countsteps+1;
   counter=counter+1;
   time = time+dt;  % Step one timestep and update time.




% Then a section where specify the boundary conditions.
% Here we specify the inlet rates of the different phases at the
% bottom of the pipe in kg/s. We interpolate to make things smooth.
% It is also possible to change the outlet boundary status of the well
% here. First we specify rates at the bottom and the pressure at the
outlet
% in case we have an open well. This is a place where we can change
the
% code to control simulations. If the well shall be close, wellopening
must
% be set to 0. It is also possible to reverse the flow (bullheading).

% In the example below, we take a gas kick and then circulate this
```

```
      % out of the well without closing the well. (how you not should perform
      % well control)

      % Note there are two variables wellopening and bullheading that can be
      % changed in the control structure below to close the well or start
      % reversing the flow i.e. pumping downwards.

      % Note that if we will change to bullheading throughout the control
stucture,
      % the variable inletligmassrate
      % has to be defined as negative since pumping downwards at outlet will
be
      % in negative direction (postive direction of flow has been chosen to
be
      % upwards)

      % NB, NOTE THAT THIS IS ONE OF THE MAIN PLACES WHERE YOU HAVE TO ADJUST
THE
      % CODE TO CONTROL THE SIMULATION SCENARIO.

      XX = 0; % Gasrate in kg/s

      YY= 2000*0.017; % Liquidrate in kg/s

      if (time < 10)

        inletligmassrate=0.0;
        inletgasmassrate=0.0;

      elseif ((time>=10) & (time < 20))
        inletligmassrate = YY*(time-10)/10;   % Interpolate the rate from 0
to value wanted.
        inletgasmassrate = XX*(time-10)/10;

      elseif ((time >=20) && (time<200))
        inletligmassrate = YY;
        inletgasmassrate = XX;
      elseif ((time >=200) & (time<210))
      %  inletligmassrate = YY-YY*(time-200)/10;
        inletligmassrate = YY-YY*(time-200)/10;
        inletgasmassrate = XX-XX*(time-200)/10;
      elseif (time > 210)
        inletligmassrate=0;
        inletgasmassrate=0;

      end

      % The commented code below are from some previous runs. It shows. e.g.
how
      % we can close the well.
      %elseif((time>=500)&(time<510))
      %   inletligmassrate = YY-YY*(time-500)/10;
      %   inletgasmassrate = XX-XX*(time-500)/10;
      % elseif(time>=510)
      %   inletligmassrate=0;
      %   inletgasmassrate=0;
      %   wellopening=0.0;
      % end
```

```
      %XX = 4;
      % XX (kg/s) is a variable for introducing a kick in the well.
      %YY = 15; % Liquid flowrate (kg/s) (1 kg/s = 1 l/s approx)
      % if (time < 10)
      %
      %   inletligmassrate=0.0;
      %   inletgasmassrate=0.0;
      %
      % elseif ((time>=10) & (time < 20))
      %   inletligmassrate = 0*(time-10)/10;
      %   inletgasmassrate = XX*(time-10)/10;
      %
      % elseif ((time >=20) & (time<110))
      %   inletligmassrate = 0;
      %   inletgasmassrate = XX;
      %
      % elseif ((time>=110)& (time<120))
      %   inletligmassrate = 0;
      %   inletgasmassrate = XX-XX*(time-110)/10;
      % elseif ((time>=120&time<130))
      %   inletligmassrate =0;
      %   inletgasmassrate =0;
      % elseif ((time>=130)&(time<300))
      %   inletligmassrate =0;
      %   inletgasmassrate =0;
      % elseif ((time>=300)&(time<310))
      %   inletligmassrate= YY*(time-300)/10;
      %   inletgasmassrate =0;
      % elseif((time>=310))
      %   inletligmassrate= YY;
      %   inletgasmassrate =0;
      % end

      kickvolume = kickvolume+inletgasmassrate/dgo(1)*dt;  % Here we find the
kickvolume

      % initially induced in the well.




      % Here we specify the physical outlet pressure. Here we have given the
pressure as
      % constant. It would be possible to adjust it during openwell conditions
      % either by giving the wanted pressure directly (in the command lines
      % above) or by finding it indirectly through a chokemodel where the
chokeopening
      % would have had to be  an input parameter. The chokeopening variable
would equally had
      % to be adjusted inside the controle structure given above.

       pressureoutlet = pbondout;

      % Based on these given physical boundary values combined with use
      % of extrapolations techniques
      % for the remaining unknowns at the boundaries, we will define the mass
and
```

```
      % momentum fluxes at the boundaries (inlet and outlet of pipe).

      % inlet/bottom fluxes first.
         if (bullheading<=0)
       % Here we pump from bottom
           flc(1,1)= inletligmassrate/area(1);
           flc(1,2)= 0.0;
           flc(1,3)= flc(1,1)*vlo(1);


           fgc(1,1)= 0.0;
           fgc(1,2)= inletgasmassrate/area(1);
           fgc(1,3)= fgc(1,2)*vgo(1);

           fp(1,1)= 0.0;
           fp(1,2)= 0.0;

      % Old way of treating the boundary
      %      fp(1,3)= po(1)+0.5*(po(1)-po(2)); %Interpolation used to find
the
      % pressure at the inlet/bottom of the well.

      % New way of treating the boundary
           fp(1,3)= po(1)...
                 +0.5*dx*(dlo(1)*evo(1)+dgo(1)*ego(1))*g...
                 +0.5*dx*fricgrad(1);



         else
           % Here we pump from the top. All masses are assumed to flow out
of the
           % well into the formation. We use first order extrapolation.
           flc(1,1)=dlo(1)*evo(1)*vlo(1);
           flc(1,2)=0.0;
           flc(1,3)=flc(1,1)*vlo(1);

           fgc(1,1)=0.0;
           fgc(1,2)=dgo(1)*ego(1)*vgo(1);
           fgc(1,3)=fgc(1,2)*vgo(1);

           fp(1,1)=0.0;
           fp(1,2)=0.0;
           fp(1,3)=20000000;  (Pa) % This was a fixed pressure set at bottom
when bullheading
         end



      % Outlet fluxes (open & closed conditions)

          if (wellopening>0.01)

      % Here open end condtions are given. We distinguish between bullheading
      % & normal circulation.
```

```
            if (bullheading<=0)  % Here we dont bullhead, i.e we circulate
from bottom

              % Here the is normal ciruclation and open well)
                flc(nofluxes,1)= dlo(nobox)*evo(nobox)*vlo(nobox);
                flc(nofluxes,2)= 0.0;
                flc(nofluxes,3)= flc(nofluxes,1)*vlo(nobox);


                fgc(nofluxes,1)= 0.0;
                fgc(nofluxes,2)= dgo(nobox)*ego(nobox)*vgo(nobox);
        %         fgc(nofluxes,2)=0; Activate if gas is sucked in!?
                fgc(nofluxes,3)= fgc(nofluxes,2)*vgo(nobox);

                fp(nofluxes,1)= 0.0;
                fp(nofluxes,2)= 0.0;
                fp(nofluxes,3)= pressureoutlet;
            else
                % Here we are bullheading.
                flc(nofluxes,1)= inletligmassrate/area(nobox);
                flc(nofluxes,2)= 0.0;
                flc(nofluxes,3)= flc(nofluxes,1)*vlo(nobox);

                fgc(nofluxes,1)=0.0;
                fgc(nofluxes,2)=0.0;
                fgc(nofluxes,3)=0.0;

                fp(nofluxes,1)=0.0;
                fp(nofluxes,2)=0.0;
                fp(nofluxes,3)= po(nobox)...
                -
0.5*dx*(dlo(nobox)*evo(nobox)+dgo(nobox)*ego(nobox))*g...
                +0.5*dx*fricgrad(nobox); %check sign here on friction
                % Physcially, the friction should be added when going from
                % mid point in upper cell to outlet. But if fricgrad(nobox)
is
                % negative there should be a minus in front of the term to
have
                % + in the end.
            end
          else

      % Here closed end conditions are given

              flc(nofluxes,1)= 0.0;
              flc(nofluxes,2)= 0.0;
              flc(nofluxes,3)= 0.0;

              fgc(nofluxes,1)= 0.0;
              fgc(nofluxes,2)= 0.0;
              fgc(nofluxes,3)= 0.0;

              fp(nofluxes,1)=0.0;
              fp(nofluxes,2)=0.0;

          %    Old way of treating the boundary
          %     fp(nofluxes,3)= po(nobox)-0.5*(po(nobox-1)-po(nobox));

          %    New way of treating the boundary
```

```
                fp(nofluxes,3)= po(nobox)...
                -0.5*dx*(dlo(nobox)*evo(nobox)+dgo(nobox)*ego(nobox))*g;
          %       -0.5*dx*fricgrad(nobox); % Neglect friction since well is
closed.
              end


        % Implementation of slopelimiters. They are applied on the physical
        % variables like phase densities, phase velocities and pressure.

      % It was found that if the slopelimiters were set to zero in
      % the boundary cells, the pressure in these became wrong. E.g. the
upper
      % cell get an interior pressure that is higher than it should be e.g.
when
      % being static (hydrostatic pressure was too high). The problem was
reduced
      % by copying the slopelimiters from the interior cells. However, both
      % approaches seems to give the same BHP pressure vs time but the latter
      % approach give a more correct pressure vs depth profile. It is also
better
      % to use when simulating pressure build up where the upper cell pressure
      % must be monitored. It should be checked more in detail before
concluding.
      % BUT; there has been mass conservation problems with the scheme for
the
      % case where the slopelimiters were copied (see master thesis of Keino)
      % A possible fix has been included below where the slopelimiter related
to
      % the gas volume fraction is set to zero in the first cell.

            for i=2:nobox-1
             sl1(i)=minmod(dlo(i-1),dlo(i),dlo(i+1),dx);
             sl2(i)=minmod(po(i-1),po(i),po(i+1),dx);
             sl3(i)=minmod(vlo(i-1),vlo(i),vlo(i+1),dx);
             sl4(i)=minmod(vgo(i-1),vgo(i),vgo(i+1),dx);
             sl5(i)=minmod(ego(i-1),ego(i),ego(i+1),dx);
             sl6(i)=minmod(dgo(i-1),dgo(i),dgo(i+1),dx);
            end

        % Slopelimiters in outlet boundary cell are set to zero!
      %      sl1(nobox)=0;
      %      sl2(nobox)=0;
      %      sl3(nobox)=0;
      %      sl4(nobox)=0;
      %      sl5(nobox)=0;
      %      sl6(nobox)=0;

        % Slopelimiters in outlet boundary cell are copied from neighbour
cell!
            sl1(nobox)=sl1(nobox-1);
            sl2(nobox)=sl2(nobox-1);
            sl3(nobox)=sl3(nobox-1);
            sl4(nobox)=sl4(nobox-1);
            sl5(nobox)=sl5(nobox-1);
            sl6(nobox)=sl6(nobox-1);

      % Slopelimiters in inlet boundary cell are set to zero!
      %      sl1(1)=0;
      %      sl2(1)=0;
```

```
%       sl3(1)=0;
%       sl4(1)=0;
%       sl5(1)=0;
%       sl6(1)=0;

% Slopelimiters in inlet boundary cell are copied from neighbour cell!
     sl1(1)=sl1(2);
     sl2(1)=sl2(2);
     sl3(1)=sl3(2);
     sl4(1)=sl4(2);
     sl5(1)=sl5(2);
     sl6(1)=sl6(2);

% FIX FOR OMITTING THE GAS MASS CONSERVATION PROBLEM
     sl5(1)=0;


% Now we will find the fluxes between the different cells.
% NB - IMPORTANE -  Note that if we change the compressibilities/sound
velocities of
% the fluids involved, we may need to do changes inside the csound
function.
% But the effect of this is unclear.

     for j = 2:nofluxes-1

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % First order method is from here: If you want to test this, activate
this
 % and comment the second order code below.
%        cl = csound(ego(j-1),po(j-1),dlo(j-1),k);
%        cr = csound(ego(j),po(j),dlo(j),k);
%        c = max(cl,cr);
%        pll = psip(vlo(j-1),c,evo(j));
%        plr = psim(vlo(j),c,evo(j-1));
%        pgl = psip(vgo(j-1),c,ego(j));
%        pgr = psim(vgo(j),c,ego(j-1));
%        vmixr = vlo(j)*evo(j)+vgo(j)*ego(j);
%        vmixl = vlo(j-1)*evo(j-1)+vgo(j-1)*ego(j-1);
%
%        pl = pp(vmixl,c);
%        pr = pm(vmixr,c);
%        mll= evo(j-1)*dlo(j-1);
%        mlr= evo(j)*dlo(j);
%        mgl= ego(j-1)*dgo(j-1);
%        mgr= ego(j)*dgo(j);
%
%        flc(j,1)= mll*pll+mlr*plr;
%        flc(j,2)= 0.0;
%        flc(j,3)= mll*pll*vlo(j-1)+mlr*plr*vlo(j);
%
%        fgc(j,1)=0.0;
%        fgc(j,2)= mgl*pgl+mgr*pgr;
%        fgc(j,3)= mgl*pgl*vgo(j-1)+mgr*pgr*vgo(j);
%
%        fp(j,1)= 0.0;
%        fp(j,2)= 0.0;
%        fp(j,3)= pl*po(j-1)+pr*po(j);
```

```
%  First order methods ends here
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Second order method starts here:
% Here slopelimiter is used on all variables except phase velocoties

       psll = po(j-1)+dx/2*sl2(j-1);
       pslr = po(j)-dx/2*sl2(j);
       dsll = dlo(j-1)+dx/2*sl1(j-1);
       dslr = dlo(j)-dx/2*sl1(j);
       dgll = dgo(j-1)+dx/2*sl6(j-1);
       dglr = dgo(j)-dx/2*sl6(j);

       vlv = vlo(j-1)+dx/2*sl3(j-1);
       vlh = vlo(j)-dx/2*sl3(j);
       vgv = vgo(j-1)+dx/2*sl4(j-1);
       vgh = vgo(j)-dx/2*sl4(j);

       gvv = ego(j-1)+dx/2*sl5(j-1);
       gvh = ego(j)-dx/2*sl5(j);
       lvv = 1-gvv;
       lvh = 1-gvh;

       cl = csound(gvv,psll,dsll,k);
       cr = csound(gvh,pslr,dslr,k);
       c = max(cl,cr);

       pll = psip(vlo(j-1),c,lvh);
       plr = psim(vlo(j),c,lvv);
       pgl = psip(vgo(j-1),c,gvh);
       pgr = psim(vgo(j),c,gvv);
       vmixr = vlo(j)*lvh+vgo(j)*gvh;
       vmixl = vlo(j-1)*lvv+vgo(j-1)*gvv;

       pl = pp(vmixl,c);
       pr = pm(vmixr,c);


      mll= lvv*dsll;
      mlr= lvh*dslr;
      mgl= gvv*dgll;
      mgr= gvh*dglr;

      flc(j,1)= mll*pll+mlr*plr;
      flc(j,2)= 0.0;
      flc(j,3)= mll*pll*vlo(j-1)+mlr*plr*vlo(j);


      fgc(j,1)=0.0;
      fgc(j,2)= mgl*pgl+mgr*pgr;
      fgc(j,3)= mgl*pgl*vgo(j-1)+mgr*pgr*vgo(j);
```

```
      fp(j,1)= 0.0;
      fp(j,2)= 0.0;
      fp(j,3)= pl*psll+pr*pslr;



%%% Second order method ends here
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Here sloplimiters is used on all variables. This
% has not worked so well yet. Therefore it is commented away.

%       psll = po(j-1)+dx/2*sl2(j-1);
%       pslr = po(j)-dx/2*sl2(j);
%       dsll = dlo(j-1)+dx/2*sl1(j-1);
%       dslr = dlo(j)-dx/2*sl1(j);
%       dgll = dgo(j-1)+dx/2*sl6(j-1);
%       dglr = dgo(j)-dx/2*sl6(j);
%
%       vlv = vlo(j-1)+dx/2*sl3(j-1);
%       vlh = vlo(j)-dx/2*sl3(j);
%       vgv = vgo(j-1)+dx/2*sl4(j-1);
%       vgh = vgo(j)-dx/2*sl4(j);
%
%       gvv = ego(j-1)+dx/2*sl5(j-1);
%       gvh = ego(j)-dx/2*sl5(j);
%       lvv = 1-gvv;
%       lvh = 1-gvh;
%
%       cl = csound(gvv,psll,dsll,k);
%       cr = csound(gvh,pslr,dslr,k);
%       c = max(cl,cr);
%
%       pll = psip(vlv,c,lvh);
%       plr = psim(vlh,c,lvv);
%       pgl = psip(vgv,c,gvh);
%       pgr = psim(vgh,c,gvv);
%       vmixr = vlh*lvh+vgh*gvh;
%       vmixl = vlv*lvv+vgv*gvv;
%
%       pl = pp(vmixl,c);
%       pr = pm(vmixr,c);
%       mll= lvv*dsll;
%       mlr= lvh*dslr;
%       mgl= gvv*dgll;
%       mgr= gvh*dglr;
%
%       flc(j,1)= mll*pll+mlr*plr;
%       flc(j,2)= 0.0;
%       flc(j,3)= mll*pll*vlv+mlr*plr*vlh;
%
%
%       fgc(j,1)=0.0;
%       fgc(j,2)= mgl*pgl+mgr*pgr;
%       fgc(j,3)= mgl*pgl*vgv+mgr*pgr*vgh;
%
%       fp(j,1)= 0.0;
%       fp(j,2)= 0.0;
```

```
%       fp(j,3)= pl*psll+pr*pslr;


    end

% Fluxes have now been calculated. We will now update the conservative
% variables in each of the numerical cells.



% The source terms can be calculated by using a
% for loop.
% Note that the model is sensitive to how we treat the model
% for low Reynolds numbers (possible discontinuity in the model)
     for j=1:nobox
      fricgrad(j)=dpfric(vlo(j),vgo(j),evo(j),ego(j),dlo(j),dgo(j),
...
        po(j),do(j),di(j),viscl,viscg); % Pa/m
      hydgrad(j)=g*(dlo(j)*evo(j)+dgo(j)*ego(j)); % Pa/m
     end

     sumfric = 0;
     sumhyd= 0;

     for j=1:nobox

    % Here we solve the three conservation laws for each cell and
update
    % the conservative variables qv

      ar = area(j);


     % Liquid mass conservation
     qv(j,1)=qvo(j,1)-dtdx*((ar*flc(j+1,1)-ar*flc(j,1))...
                           +(ar*fgc(j+1,1)-ar*fgc(j,1))...
                           +(ar*fp(j+1,1)-ar*fp(j,1)));

     % Gas mass conservation:

     qv(j,2)=qvo(j,2)-dtdx*((ar*flc(j+1,2)-ar*flc(j,2))...
                           +(ar*fgc(j+1,2)-ar*fgc(j,2))...
                           +(ar*fp(j+1,2)-ar*fp(j,2)));
     % Mixture momentum conservation:

     qv(j,3)=qvo(j,3)-dtdx*((ar*flc(j+1,3)-ar*flc(j,3))...
                           +(ar*fgc(j+1,3)-ar*fgc(j,3))...
                           +(ar*fp(j+1,3)-ar*fp(j,3)))...
                 -dt*ar*(fricgrad(j)+hydgrad(j));

%  Add up the hydrostatic pressure  and friction  in the whole well.
     sumfric=sumfric+fricgrad(j)*dx;
     sumhyd=sumhyd+hydgrad(j)*dx;

     end
```

```
      % Section where we find the physical variables (pressures, densities
etc)
      % from the conservative variables. Some trickes to ensure stability.
These
      % are induced to avoid negative masses.



          gasmass=0;
          liqmass=0;

          for j=1:nobox


      % Remove the area from the conservative variables to find the
      % the primitive variables from the conservative ones.

           qv(j,1)= qv(j,1)/area(j);
           qv(j,2)= qv(j,2)/area(j);

           if (qv(j,1)<0.00000001)  % Trick to avoid negative masses.
             qv(j,1)=0.00000001;
           end

           if (qv(j,2)< 0.00000001)  % Trick to avoid negative masses.
             qv(j,2)=0.00000001;
           end

       % Here we summarize the mass of gas and liquid in the well
respectively.
       % These variables are important to show that the scheme is conserving
       % mass. (if e.g. gas leaks in our out of the well unintentionally in
the simulation
       % without being specified in the code,something fundamental is wrong.

           gasmass = gasmass+qv(j,2)*area(j)*dx;
           liqmass = liqmass+qv(j,1)*area(j)*dx;

      % Below, we find the primitive variables pressure and densities based
on
      % the conservative variables q1,q2. One can choose between getting them
by
      % analytical or numerical solution approach specified in the beginning
of
      % the program. Ps. For more advanced density models, this must be
changed.


           if (analytical == 1)
      %       % Analytical solution:

      % here the simple density models used in PET 510 Wellflow modelling
      % compendium is used.

            t1=rho0-P0/al^2;
```

```
      %  Coefficients:
           a = 1/(al*al);
           b = t1-qv(j,1)-rt*qv(j,2)/(al*al);
           c = -1.0*t1*rt*qv(j,2);
      %

      %       Note here we use the very simple models from the PET510 course
              p(j)=(-b+sqrt(b*b-4*a*c))/(2*a);  % Pressure
              dl(j)=rholiq(p(j),temp(j)); % Density of liquid
              dg(j)=rogas(p(j),temp(j)); % Density of gas

      %     The code below can be activated if we want to switch to the other
set
      %     of density models. Also then remember to do the changes inside
      %     functions rogas og rholiq if we change density models.

      %           x1=rho0-P0*rho0/Bheta-rho0*Alpha*(temp(j)-T0);
      %           x2=rho0/Bheta;
      %           x3=-qv(j,2)*R*temp(j);

      %           a = x2;
      %           b = x1+x2*x3-qv(j,1);
      %           c = x1*x3;

      %           p(j)=(-b+sqrt(b*b-4*a*c))/(2*a);  % Pressure
      %           dl(j)=rholiq(p(j),temp(j));
      %           dg(j)=rogas(p(j),temp(j));
            else


            %Numerical Solution: This might be used if we use more complex
            %density models. Has not been used for years.

             [p(j),error]=itsolver(po(j),qv(j,1),qv(j,2)); % Pressure
             dl(j)=rholiq(p(j),temp(j)); % Density of liquid
             dg(j)=rogas(p(j)); % Density of gas

            % Incase a numerical solution is not found, the program will
write out "error":
             if error > 0
                error
             end
            end

      %   Find phase volume fractions
            eg(j)= qv(j,2)/dg(j);
            ev(j)=1-eg(j);

       %    Reset average conservative varibles in cells with area included
in the variables.

           qv(j,1)=qv(j,1)*area(j);
           qv(j,2)=qv(j,2)*area(j);

           end  % end of loop
```

```
        %    Below we find the phase velocities by combining the
        %    conservative variable defined by the mixture momentum equation
        %    with the gas slip relation.
        %    At the same time we try to summarize the gas volume in the well.
This
        %    also measure the size of the kick.


          gasvol=0;

          for j=1:nobox


          % The  interpolations introduced below are included
          % to omit a singularity in the slip relation when the gas volume
          % fraction becomes equal to 1/K. In additon, S is interpolated to
          % zero when approaching one phase gas flow. In the transition to
          % one phase gas flow, we have no slip condtions (K=1, S=0)
          % We will let the k0,s0,k1,s1 be arrays to make it easier to
incorporate
          % different flow regimes later.  In that case, the slip parameters
will
          % vary from cell to cell and we must have slip parameter values for
each
          % cell.

             ktemp=k;
             stemp=s;

             k0(j) = ktemp;
             s0(j) = stemp;

           % Interpolation to handle that (1-Kxgasvolumefraction) does not
become zero
             if ((eg(j)>=0.7) & (eg(j)<=0.8))
               xint = (eg(j)-0.7)/0.1;
               k0(j) =1.0*xint+k*(1-xint);
             elseif(eg(j)>0.8)
               k0(j)=1.0;
             end


           % Interpolate S to zero in transition to pure gas phase
              if ((eg(j)>=0.9) & (eg(j)<=1.0))
                xint = (eg(j)-0.9)/0.1;
                s0(j) = 0.0*xint+s*(1-xint);
              end

         % Note that the interpolations above and below can be changed
         % if numerical stability problems
         % are encountered.

         %
             if (eg(j)>=0.999999)
```

```
            % Pure gas
              k1(j) = 1.0;
              s1(j) = 0.0;
            else
              %Two phase flow
              k1(j) = (1-k0(j)*eg(j))/(1-eg(j));
              s1(j) = -1.0*s0(j)*eg(j)/(1-eg(j));
            end



            help1 = dl(j)*ev(j)*k1+dg(j)*eg(j)*k0;
            help2 = dl(j)*ev(j)*s1+dg(j)*eg(j)*s0;


            vmixhelpl = (qv(j,3)/area(j)-help2)/help1;
            vg(j)=k0(j)*vmixhelpl+s0(j);
            vl(j)=k1(j)*vmixhelpl+s1(j);




            % Variable for summarizing the gas volume content in the well.
            gasvol=gasvol+eg(j)*area(j)*dx;


          end


        % Old values are now set equal to new values in order to prepare
        % computation of next time level.


          po=p;
          dlo=dl;
          dgo=dg;
          vlo=vl;
          vgo=vg;
          ego=eg;
          evo=ev;
          qvo=qv;



        % Section where we save some timedependent variables in arrays.
        % e.g. the bottomhole pressure. They will be saved for certain
        % timeintervalls defined in the start of the program in order to ensure
        % that the arrays do not get to long!

          if (counter>=nostepsbeforesavingtimedata)
            printcounter=printcounter+1;
            time  % Write time to screen.

            % Outlet massrates (kg/s) vs time

liquidmassrateout(printcounter)=dl(nobox)*ev(nobox)*vl(nobox)*area(nobox);
```

```
gasmassrateout(printcounter)=dg(nobox)*eg(nobox)*vg(nobox)*area(nobox);

          % Outlet flowrates (lpm) vs time

liquidflowrateout(printcounter)=liquidmassrateout(printcounter)/...
              rholiq(P0,T0)*1000*60;
          gasflowrateout(printcounter)=gasmassrateout(printcounter)/...
              rogas(P0,T0)*1000*60;

          % Hydrostatic and friction pressure (bar) in well vs time
          hyd(printcounter)=sumhyd/100000;
          fric(printcounter)=sumfric/100000;

          % Volume of gas in well vs time (m3). Also used for indicating kick
          % size in well.

          volgas(printcounter)=gasvol;

          % Total phase masses (kg) in the well vs time
          % Used for checking mass conservation.

           massgas(printcounter)=gasmass;
           massliq(printcounter)=liqmass;


          % pout calculates the pressure at the outletboundary. I.e. upper
edge
          % of uppermost cell. Corresponds where the well ends at surface.
The
          % reason we do this is the fact than in AUSMV is all variables
defined
          % in the mid point of the numerical cells.
          pout(printcounter)=(p(nobox)-0.5*dx*...
          (dlo(nobox)*evo(nobox)+dgo(nobox)*ego(nobox))*g-
dx*0.5*fricgrad(nobox))/100000;


          % pin (bar) defines the  pressure at the inlet boundary, I.e lower
edge
          % of the lowermost cell. Corresponds to TD of well.
          pin(printcounter)=
(p(1)+0.5*dx*(dlo(1)*evo(1)+dgo(1)*ego(1))*g+0.5*dx*fricgrad(1))/100000;

          % Pressure in the middle of top box (bar).
          pnobox(printcounter)=p(nobox)/100000;  %



          % Time variable
          timeplot(printcounter)=time;

          counter = 0;


        end
      end
```

```
% end of stepping forward in time.



% Printing of resultssection


countsteps % Marks number of simulation steps.


% Plot commands for variables vs time. The commands can also
% be copied to command screen where program is run for plotting other
% variables.

toc,
e = cputime-t

% Plot bottomhole pressure
plot(timeplot,pin)

% Show cfl number used.
disp('cfl')
cfl = al*dt/dx

 plot(timeplot,pin)
%plot(timeplot,hyd)
%plot(timeplot,fric)
%plot(timeplot,liquidmassrateout)
%plot(timeplot,gasmassrateout)
%plot(timeplot,volgas)
%plot(timeplot,liquidflowrateout)
%plot(timeplot,gasflowrateout)
%plot(timeplot,massgas)
%plot(timeplot,massliq)
%plot(timeplot,pout)
%plot(timeplot,pnobox)

%Plot commands for variables vs depth/Only the last simulated
%values at endtime is visualised

%plot(vl,x);
%plot(vg,x);
%plot(eg,x);
%plot(p,x);
%plot(dl,x);
%plot(dg,x);
```

B.2 csound.m

```
function mixsoundvelocity = csound(gvo,po,dlo,k)
% Note that at this time k is set to 1.0 (should maybe be
% included below

temp= gvo*dlo*(1.0-gvo);
a=1;
```

```
if (temp < 0.01)
  temp = 0.01;
end

cexpr = sqrt(po/temp);

if (gvo <= 0.5)
 mixsoundvelocity = min(cexpr,1500);
else
 mixsoundvelocity = min(cexpr,316);
end


%mixsoundvelocity = 1500*(1-gvo)+6000*gvo;
```

B.3 dpfric.m

```
function                              friclossgrad                              =
dpfric(vlo,vgo,evo,ego,dlo,dgo,pressure,do,di,viscl,viscg)

%friclossgrad =
%dpfric(vlo,vgo,evo,ego,dlo,dgo,pressure,do,di,viscl,viscg)
% Works for two phase flow. The one phase flow model is used but mixture
 % values are introduced.

%  rhol = dlo;
%  rhog = dgo;
%  vmixfric = vlo.*evo+vgo.*ego;
%  viscmix =  viscl.*evo+viscg.*ego;
%  densmix = dlo.*evo+dgo.*ego;
%
%  % Calculate mix reynolds number
%  Re = ((densmix.*abs(vmixfric).*(do-di))./viscmix);
%
%  % Calculate friction factor. For Re > 3000, the flow is turbulent.
%  % For Re < 2000, the flow is laminar. Interpolate in between.
%
%  if (Re<0.001)
%    f=0.0;
%  else
%   if (Re >= 3000)
%    f = 0.052*Re.^(-0.19);
%   elseif ( (Re<3000) & (Re > 2000))
%    f1 = 24./Re;
%    f2 = 0.052*Re.^(-0.19);
%    xint = (Re-2000)./1000.0;
%    f = (1.0-xint).*f1+xint.*f2;
%   else
%    f = 24./Re;
%   end
%  end
%
%   friclossgrad = ((2*f.*densmix.*vmixfric.*abs(vmixfric))./(do-di));


 vmixfric = vlo*evo+vgo*ego;
 viscmix =  viscl*evo+viscg*ego;
```

```
densmix = dlo*evo+dgo*ego;

% Calculate mix reynolds number
Re = ((densmix*abs(vmixfric)*(do-di))/viscmix);

% Calculate friction factor. For Re > 3000, the flow is turbulent.
% For Re < 2000, the flow is laminar. Interpolate in between.

if (Re<0.001)
  f=0.0;
else
 if (Re >= 3000)
  f = 0.052*Re^(-0.19);
 elseif ( (Re<3000) & (Re > 2000))
  f1 = 24/Re;
  f2 = 0.052*Re^(-0.19);
  xint = (Re-2000)/1000.0;
  f = (1.0-xint)*f1+xint*f2;
 else
  f = 24/Re;
 end
end

% if (Re<100)
%  f = 0.0;
% else
%   if (Re<200)
%    f1 = 0;
%    f2 = 24/Re;
%    xint = (Re-0)/200;
%    f = (1-xint)*f1+xint*f2;
%
%   elseif ((Re>=200)&(Re<2000))
%    f = 24/Re;
%
%   elseif ((Re>=2000)&(Re<3000))
%    f1 = 24/Re;
%    f2 = 0.052*Re^(-0.19);
%    xint = (Re-2000)/1000.0;
%    f = (1.0-xint)*f1+xint*f2;
%   else
%    f = 0.052*Re^(-0.19);
%   end
% end



  friclossgrad = ((2*f*densmix*vmixfric*abs(vmixfric))/(do-di));

%     if (friclossgrad <0)
%       friclossgrad = 0;
%     end
end
```

B.4 minmod.m

```
function [ slope ] = minmod(x1,x2,x3,dx)
```

```
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

a = x2-x1;
b = x3-x2;

if (a*b)<=0
 slope = 0;
else
  if (abs(a)<abs(b))
   slope = a;
  else
   slope = b;
  end
end

slope = slope/dx;


end
```

B.5 pm.m

```
function pmvalue = pm(v,c)


  if (abs(v)<=c)
    pmvalue = -1.0*(v-c)*(v-c)/(4*c)*(-2.0-v/c)/c;
  else
    pmvalue = 0.5*(v-abs(v))/v;
  end
end
```

B.6 pp.m

```
function pmvalue = pp(v,c)

  if (abs(v)<=c)
    pmvalue = (v+c)*(v+c)/(4*c)*(2.0-v/c)/c;
  else
    pmvalue = 0.5*(v+abs(v))/v;
  end
end
```

B.7 psim.m

```
function pmvalue = psim(v,c,alpha)

  if (abs(v)<=c)
    pmvalue = -1.0*alpha*(v-c)*(v-c)/(4*c)+(1-alpha)*(v-abs(v))/2;
  else
    pmvalue = 0.5*(v-abs(v));
  end
end
```

B.8 psip.m

```
function pmvalue = psip(v,c,alpha)


  if (abs(v)<=c)
    pmvalue = alpha*(v+c)*(v+c)/(4*c)+(1-alpha)*(v+abs(v))/2;
  else
    pmvalue = 0.5*(v+abs(v));
  end
end
```

B.9 rholiq.m

```
function [rhol] = rholiq(pressure,temperature)
%Simple model for liquid density
% p0 = 100000.0; % Assumed
% t0 = 20+273.15;
%
% beta = 2.2*10^9;
% alpha = 0.000207;
% rho0 = 1000;
%
% %rhol = 1000.0 + (pressure-p0)/(1500.0*1500.0);
%  rhol  =  rho0+((rho0/beta)*(pressure-p0))-(rho0*alpha*(temperature-
t0));

% SIMPLE PET 510 Model below:

if (pressure < 100000)
 pressure = 100000;
end

rhol = 1000+ (pressure-100000)/1500^2;
end
```

B.10 rogas.m

```
function rhog = rogas(pressure,temp)

%Simple gas density model. Temperature is neglected.
% rhogas = pressure / (velocity of sound in the gas phase)^2 = pressure
/
% rT --> gas sound velcoity = SQRT(rT)

 % rhog = 4200;
%  R = 286.9;
%  rhog = pressure/(R*temp);

% SIMPLE PET 510 model below:

if (pressure < 100000)
 pressure = 100000;
end

rhog = pressure/100000;
```

Appendix C New Code

C.1 Tw_NarrowSlotNewtonianLaminar.m

```
function [Tw] = Tw_NarrowSlotPowerLawLaminar(Flowrate, VelocityPipe, GeometryPar,
FluidParameters)
% This function calculates the narrow slot approximation
% for Tw using Newtonian laminar flow

    mu = FluidParameters.Viscosity.mu;
    AreaCrs = GeometryPar.AreaCrs;
    velocityFluid = Flowrate ./ AreaCrs - VelocityPipe;
    h = GeometryPar.h;


    Tw = -6 * mu ./ h .* velocityFluid;
end
```

C.2 Tw_NarrowSlotPowerLawLaminar.m

```
function [Tw] = Tw_NarrowSlotPowerLawLaminar(Flowrate, VelocityPipe,
GeometryPar, FluidParameters)
% This function calculates the narrow slot approximation
% for Tw using power-law laminar


    mu = FluidParameters.Viscosity.mu;
    AreaCrs = GeometryPar.AreaCrs;
    velocityFluid = Flowrate ./ AreaCrs - VelocityPipe;
    h = GeometryPar.h;
    k = FluidParameters.Viscosity.ConsistencyIndex;
    n = FluidParameters.Viscosity.FlowBehaviorIndex;

    Tw = -abs(k.*((4*n+2)./(n*h).*abs(velocityFluid)).^n);

end
```

C.3 Tw_NarrowSlotBinghamPlasticLaminar.m

```
function [Tw] = Tw_NarrowSlotBinghamPlasticLaminar(Flowrate, VelocityPipe,
GeometryPar, FluidParameters)
% This function calculates the narrow slot approximation
% for Tw using power-law laminar

    Ty = FluidParameters.Viscosity.YieldPoint;
    mu = FluidParameters.Viscosity.mu;
    L = GeometryPar.LengthPrGrid;
    h = GeometryPar.h;
    k = FluidParameters.Viscosity.ConsistencyIndex;
    n = FluidParameters.Viscosity.FlowBehaviorIndex;
    AreaCrs = GeometryPar.AreaCrs;
    velocityFluid = Flowrate ./ AreaCrs - VelocityPipe;
```

```
    % Simplified bingham plastic for initial guess
    Tw_initial_guess = -abs(Ty + mu*6.*Flowrate./(h^3));

    if abs(Tw_initial_guess) > Ty
        % Define the equation to be solved
        eqn = @(Tw) velocityFluid + h./(6*mu).*Tw.*(1-
3/2*Ty./abs(Tw)+1/2*(Ty./abs(Tw)).^3);

        % Use fsolve to find the solution
        options = optimoptions('fsolve','Display','off');
        Tw = fsolve(eqn, Tw_initial_guess, options);
    else
        Tw = Tw_initial_guess;
    end

end
```

C.4 Tw_NarrowSlotHerschelBulkley.m

```
function [Tw] = Tw_NarrowSlotHerschelBulkleyLaminar(Flowrate, VelocityPipe,
GeometryPar, FluidParameters)
% This function calculates the narrow slot approximation
% for Tw using Herschel Bulkley laminar


    Ty = FluidParameters.Viscosity.YieldPoint;
    mu = FluidParameters.Viscosity.mu;
    L = GeometryPar.LengthPrGrid;
    h = GeometryPar.h;
    k = FluidParameters.Viscosity.ConsistencyIndex;
    n = FluidParameters.Viscosity.FlowBehaviorIndex;
    AreaCrs = GeometryPar.AreaCrs;
    velocityFluid = Flowrate ./ AreaCrs - VelocityPipe;

    % Simplified bingham plastic for initial guess
    Tw_initial_guess = -abs(Ty + mu*6.*Flowrate./(h^3));

    if abs(Tw_initial_guess) > Ty
        try
            % Define the equation to be solved
            eqn = @(Tw) velocityFluid + h./(2*k.^n.*Tw).*(abs(Tw)-
Ty).^(n+1)./(n+1).*(1-(abs(Tw)-Ty)./((n+2).*abs(Tw)));

            % Use fsolve to find the solution
            options = optimoptions('fsolve','Display','off');
            Tw = fsolve(eqn, Tw_initial_guess, options);
        catch
            Tw = Tw_initial_guess;
        end
    else
        Tw = Tw_initial_guess;
    end
```

```
end
```

C.5 TrainingDataCollector.m

```
% Define the ranges for the parameters
tic
variations = 10;
flowrate_range = linspace(0, 3000/60000, 40); % convert lpm to m^3/s
Ty_range = linspace(0, 10, variations);
mu_range = linspace(0.2, 0.6, variations);
n_range = 0.5; %linspace(0.5, 1, variations);
k_range = 0.5; %linspace(0.1, 0.5, variations);
L_range = linspace(2000, 6000, variations);
d_outer_range = linspace(7.5*0.0254, 15*0.0254, variations); % convert inches to
meters
failures = 0;
attempts = 0;
% Iteration counter to track progress
total_iterations = length(flowrate_range) * length(Ty_range) * ...
    length(mu_range) * length(n_range)* length(k_range) ...
    * length(d_outer_range) * length(L_range);
% Initialize arrays to store the inputs and outputs
inputs = [];
outputs = [];

% Loop over all combinations of parameters
for flowrate = flowrate_range
    for Ty = Ty_range
        for mu = mu_range
            for n = n_range
                for k = k_range
                    for d_outer = d_outer_range
                        for L = L_range
                            attempts = attempts + 1;
                            % Calculate the remaining parameters
                            FluidParameters.Viscosity.YieldPoint = Ty;
                            FluidParameters.Viscosity.mu = mu;
                            FluidParameters.Viscosity.FlowBehaviorIndex = n;
                            FluidParameters.Viscosity.ConsistencyIndex = k;
                            GeometryPar.LengthPrGrid = L;

                            d = 5*0.0254; % convert inches to meters for a
constand inner diameter of 5 inches
                            GeometryPar.h = d_outer - d;
                            R_inner = d/2;
                            R_outer = d_outer/2;
                            GeometryPar.AreaCrs = pi * (R_outer^2 - R_inner^2);

                            % Calculate the output
                            try
                                Tw = Tw_NarrowSlotBinghamPlasticWithFlow(flowrate,
0, GeometryPar, FluidParameters);
```

```
                                % Append the input parameters and
                                % output to the arrays
                                inputs = [inputs; [flowrate, Ty, mu,
GeometryPar.h]];%n, k, GeometryPar.h]];
                                outputs = [outputs; Tw];

                            catch
                                failures = failures + 1;
                            end
                        end
                    end
                end
            end
        end
        percentage = attempts / total_iterations * 100;
        fprintf('Simulating %.2f%%\n', percentage);
    end
    toc
end
disp(size(inputs));
disp(size(outputs));
attempts
failures
ValidResults = attempts-failures
% Save the inputs and outputs to a .mat file
save('training_dataBPFlow.mat', 'inputs', 'outputs');
toc
```

C.6 TrainingModel.m

```
% Load the data
load('training_dataBPFlow.mat')

% Split the data into a training set and a test set
m = size(inputs, 1);
idx = randperm(m);
mTrain = floor(0.7 * m);
trainIdx = idx(1:mTrain);
testIdx = idx(mTrain+1:end);

inputsTrain = inputs(trainIdx, :);
outputsTrain = outputs(trainIdx, 1);
inputsTest = inputs(testIdx, :);
outputsTest = outputs(testIdx, 1);

% Train the Random Forest model
numTrees = 100; % Number of trees in the forest
mdl = TreeBagger(numTrees, inputsTrain, outputsTrain, 'Method', 'regression',
'OOBPrediction', 'on');
% Print out-of-bag error over the number of grown trees
oobErrorBaggedEnsemble = oobError(mdl);
figure;
plot(oobErrorBaggedEnsemble);
xlabel 'Number of grown trees';
ylabel 'Out-of-bag classification error';

% Save the trained model
```

```
save('trainedModelBPFlow.mat', 'mdl');
```

C.7 Tw_NarrowSlotBPWithFlowML.m

```
function [Tw] = Tw_NarrowSlotBinghamPlasticBPWithFlowML(Flowrate, VelocityPipe,
GeometryPar, FluidParameters)
% This function calculates the narrow slot approximation
% for Tw using bingham plastic laminar based machine learning

    Ty = FluidParameters.Viscosity.YieldPoint;
    mu = FluidParameters.Viscosity.mu;
    L = GeometryPar.LengthPrGrid;
    h = GeometryPar.h;
    k = FluidParameters.Viscosity.ConsistencyIndex;
    n = FluidParameters.Viscosity.FlowBehaviorIndex;
    AreaCrs = GeometryPar.AreaCrs;
    velocityFluid = Flowrate ./ AreaCrs - VelocityPipe;
    Tw = [];
    Result = [];
    StopFlow = ones(1,length(Flowrate));
    global trainedModel
    mdl = trainedModel;
    % Use the model to make predictions on new data
    for i = 1:length(Ty)
        newInputs = [Flowrate(i), Ty(i), mu(i), h];% n(i), k(i), h];
        Result = predict(mdl, newInputs);
        Tw = [Tw, Result(1)];
        if Result <= Ty(i) && i > 1
            StopFlow(i-1) = 0;
        end
    end
    Tw = [Tw, StopFlow];
end
```

C.8 TestMLvsCalculation.m

```
% Testing of machine learning vs calculation
clear
clc

FluidParameters.Viscosity.YieldPoint = 1;
FluidParameters.Viscosity.mu = 0.5;
GeometryPar.LengthPrGrid = 1000;
GeometryPar.h = 0.204;
FluidParameters.Viscosity.ConsistencyIndex = 0.2;
FluidParameters.Viscosity.FlowBehaviorIndex = 0.8;
di = 0.127; % Inner diameter, 0.127 meters = 5 inches
do = 0.331; % Outer diameter, 0.331 meters ~ 13 inches
GeometryPar.AreaCrs = pi * (do^2 - di^2) / 4;
VelocityPipe = 0;
TwMachineLearning = [];
TwCalculation = [];
```

```
Flowrate = [];

global trainedModel;
load('trainedModelBPFlow.mat');
trainedModel = mdl;

for i = 1:1000
    Flowrate = [Flowrate , 1/(30*(1+exp(-0.01*(i-500))))];
end
tic
for i = 1:1000
    TwMachineLearning = [TwMachineLearning;
Tw_NarrowSlotBPWithFlowML(Flowrate(i)*ones(4), VelocityPipe, GeometryPar,
FluidParameters)];
    i
end
toc
tic
for i = 1:1000
    TwCalculation = [TwCalculation;
Tw_NarrowSlotBinghamPlasticWithFlow(Flowrate(i), VelocityPipe, GeometryPar,
FluidParameters)];
end
toc
RMSE = 0;
for i = 1:length(TwCalculation)
    RMSE = RMSE + (TwCalculation(i)-TwMachineLearning(i))^2;
end
RMSE = sqrt(1/length(TwMachineLearning)*RMSE)

% Plot the vectors
x = linspace(0, 999, 1000);
disp(length(x));
disp(length(TwCalculation(:, 1)));
figure; % This creates a new figure window
plot(x, TwCalculation(:, 1)); % This plots y1 versus x

hold on; % This allows the next plot to be overlaid on the same figure

plot(x, TwMachineLearning(:,1),'--r'); % This plots y2 versus x

save('RandomForestResults.mat', 'TwMachineLearning', 'TwCalculation');

% Add labels and title
xlabel('i');
ylabel('Tw');
```

Appendix D Changelog for Alf Kristian Gjerstad Model

D.1 PipeFluGen_2xOrd_Init.m

Added at line 80 to use annular space, while commenting out the similar section used for pipe from line 76 to line 78.

```
% Geometry vector parameters - Annulus (Vegard)
di = 0.127; % Inner diameter, 0.127 meters = 5 inches
do = 0.331; % Outer diameter, 0.331 meters ~ 13 inches
p.AreaCrs   = pi * (do^2 - di^2) / 4;
p.AreaSrf   = pi * (do + di) .* p.LengthPrGrid;
p.Volume    = p.AreaCrs .* p.LengthPrGrid;
p.h         = (do - di)/2;  % h brukt for narrow slot approximation
```

D.2 DsMain_Horizontal_2xOrd_ODE

Added to line 129 to bring h from PipeFluGen_2xOrd_Init.m to the different Tw calculation functions

```
GeometryPar.h   = p.h;
```

Added to Line 134 for the use of shear stress calculation functions

```
        switch c
            case 1
                wallshearStressPipeLaminar = Tw_stringNewtonianLaminar(FlowCms,
VelocityPipe, GeometryPar, p.Fluid);
            case 2
                wallshearStressPipeLaminar =
Tw_NarrowSlotNewtonianLaminar(FlowCms, VelocityPipe, GeometryPar, p.Fluid);
            case 3
                wallshearStressPipeLaminar = Tw_NarrowSlotPowerLawLaminar(FlowCms,
VelocityPipe, GeometryPar, p.Fluid);
            case 4
                wallshearStressPipeLaminar =
Tw_NarrowSlotBinghamPlasticLaminar(FlowCms, VelocityPipe, GeometryPar, p.Fluid);
            case 5
                wallshearStressPipeLaminar = Tw_NarrowSlotHerschelBulkley(FlowCms,
VelocityPipe, GeometryPar, p.Fluid);
            case 6
                wallshearStressPipeLaminar =
Tw_NarrowSlotBinghamPlasticMachineLearning(FlowCms, VelocityPipe, GeometryPar,
p.Fluid);
            case 7
                % shear stress functions that stop flow when Tw <= Ty
                wallshearStressPipeLaminar =
Tw_NarrowSlotBinghamPlasticWithFlow(FlowCms, VelocityPipe, GeometryPar, p.Fluid);
        end
```

D.3 MasterAlg_PipeHorizontal.m

Added to line 29 to allow for preloading of the machine learning model.

```
% Loading Machine Learning model
global trainedModel
load('trainedModelBPFlow.mat');
trainedModel = mdl;
```