University of Southern Queensland

Faculty of Health, Engineering & Sciences

Apply Advanced Process Control to Fine Screening Circuit in Large Mineral Processing Plant

A dissertation submitted by

A. Kapor

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Instrumentation, Control and Automation Engineering

Submitted: October, 2021

Abstract

This research project focuses on improving the control of a fine-ore screening circuit in a large gold processing plant. At this plant, ore is classified across a set of eight vibrating screens in which the undersize material reports to four ball mills and the oversize material is returned for further crushing. While the feed-rates to each individual fine screen and the total oversize return is known, there is a lack of instrumentation to reliably measure individual screen oversize. Inadequate classification of ore is an often-overlooked cause of poor mill performance (Rotich et al. 2016, p. 3889) and by obtaining a better understanding of individual screen performance, the impact on production can be mitigated.

Through the development of an appropriate model, a parameter estimation problem is derived where the efficiency of the eight vibrating screens can be estimated via regression of historical data, allowing the screen oversize to be predicted. The performance of multiple algorithms for parameter estimation were then assessed in terms of accuracy, speed and repeatability. The most appropriate algorithm was selected to form the basis of an online estimation program, developed in MATLAB®. The fastest algorithm was a simple least squares fit, though a quadratic programming method was ultimately selected as it allowed bounds to be set on the parameters.

While the implementation performs well over longer timeframes it can be unreliable at shorter intervals when variance in feed is low. Further modification to the algorithm and the introduction of regular system excitation is recommended. A further result of the analysis is the development an additional algorithm to maximise the mill feed based on the oversize model output. This research demonstrates how parameter estimation techniques can be employed using historical and real-time data to form inferential measurement points. While the model needs some refinement, the approach proved to be a viable method with future applicability.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

A. Kapor

Acknowledgments

My greatest gratitude goes to my wife, Ruby. None of this would have been possible without her unwavering support over the last eight years. Secondly, to my three children, Indiana, Theodore and Charlotte, who have I have watched grow up over this time remaining ever-patient with my studies.

I would also like to acknowledge my USQ supervisor, Catherine Hills, for her support, guidance and feedback throughout what has been a challenging time.

A. Kapor

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ANN	Artificial Neural Network	17
APC	Advanced Process Control	2
CV	Coefficient of Variation	60
DCS	Distributed Control System	xvi
DE	Differential Evolution	16
FOPDT	First Order Plus Dead Time	30
FSOS	Fine Screen Oversize Solver	xvi
GA	Genetic Algorithm	15
GUI	Graphical User Interface	xv
HPGR	High Pressure Grinding Roll	2
HMI	Human Machine Interface	20
IMC	Internal Model Controller	16
MISO	Multiple Input Single Output	30
OLS	Ordinary Least Squares	13
OPC	Open Platform Communications	20
OPC-DA	Open Platform Communications - Data Access	20
OPC-UA	Open Platform Communications - Unified Access	21
OOS	Out-Of-Sample	19
PID	Proportional, Integral & Derivative	16
PSO	Particle Swarm Optimisation	15
PLC	Programmable Logic Control	20
QP	Quadratic Programming	15

Chapter 1

Introduction

This research project presents a real parameter estimation problem in the form of a fine ore screening and conveying system where the instantaneous efficiency of eight vibrating screens needs to be determined from existing data. Inadequate classification of ore is an often overlooked cause of poor mill performance (Rotich et al. 2016, p. 3889) and by obtaining a better understanding of individual screen performance, the impact on production can be mitigated. In this respect, the absolute accuracy of the screen efficiency estimates is not as important as its repeatability, which will allow the comparison between each screen to be made, as well as a comparison with some baseline value. Through the development of a linear system model, and the continuous estimation of its parameters, it is expected that a model based optimising control strategy can be implemented, increasing throughput and driving other decision making processes.

This dissertation will first provide an introduction to the plant in which this project is based, and give an outline of the process streams that form the focus of the research. It will be followed by an explanation of the challenges and opportunities that ultimately motivated the subsequent objectives of this research. Chapter 2 then discusses the literature that was found to be relevant to the theoretical design and the implementation of the proposed solution, starting with the general theory of ore screening and finishing with concepts concerned with the software development. Chapter 3, forms the main body of the work, and discusses the theoretical formulation of the system model, as well the selection of algorithms and design of the system architecture. Chapter 4 discusses the practical aspects of the implementation, provides an analysis of its efficacy gathered from

testing and simulation, and demonstrates how the model outputs may be of benefit the process control strategy. Finally, the conclusions presented in Chapter 5, compare the outcomes achieved with those set out at the beginning of the project, and presents some opportunities for further work and improvement.

1.1 Boddington Gold Mine

Newmont's Boddington site is a gold and copper mine situated in the south-west of Western Australia approximately 120 km from Perth. With an attributable gold production of over 700 thousand ounces per year, the operation currently processes approximately 40 million metric tonnes of ore annually through a plant that includes crushing, screening, milling, copper flotation and gold recovery operations (Newmont Corporation 2021). Control of the plant equipment is primarily performed with a Yokogawa DCS augmented with many Advanced Process Control (APC) strategies, including expert systems optimising the secondary crushing, milling and flotation circuits. The plant has recently undergone significant upgrades to crushing area conveyor capacities, highlighting new bottlenecks and new opportunities for optimisation.

1.2 Fine Screening Process

The focus of this research is on the fine screening process at Boddington that removes oversize material from the fine ore-stream before reaching the ball mills. Prior to this, coarse ore is crushed by four parallel High Pressure Grinding Rolls (HPGRs) that form the the tertiary crushing plant. The crushed ore is then transported by conveyor to one of eight fine ore bins. As can be seen from the photograph in Figure 1.1, the four milling trains are then fed from the fine ore bins through a pair of banana screens (Figures 1.2, 1.4) for each, with the oversize material reporting back to the tertiary crushing plant via the return conveyor pictured in Figure 1.5.

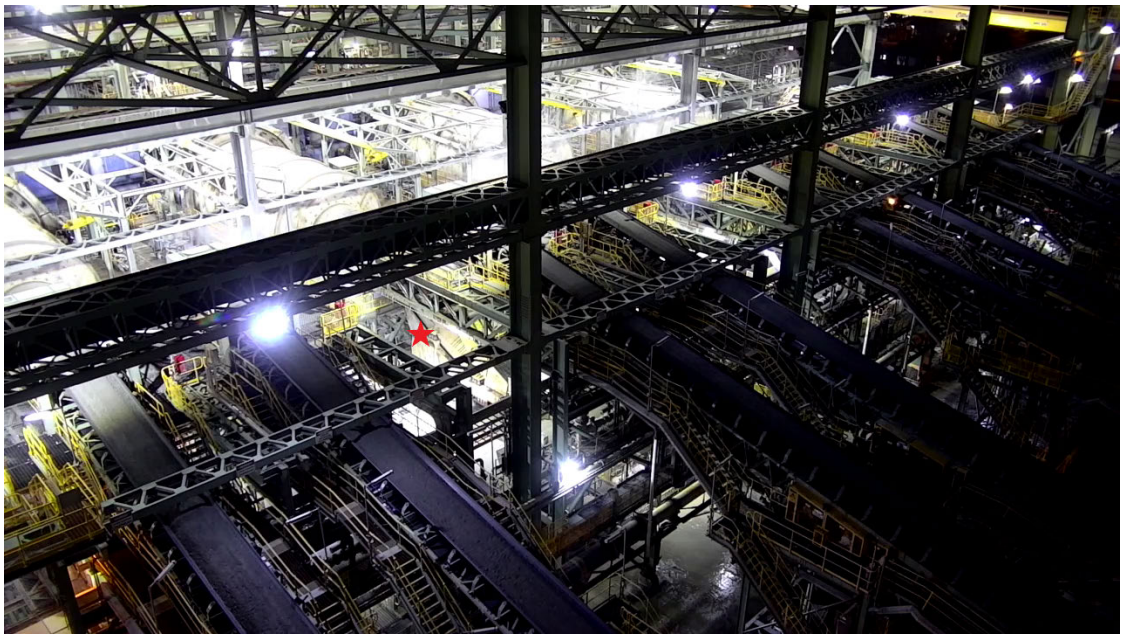

Figure 1.1: Top-down view of mill feeders (foreground) as they feed onto fine screens (red star) prior to feeding the ball mills (background).

While the size distribution in the feed is wholly affected by upstream crushing processes, this study is mainly concerned with the fine screens themselves, the feed to them, and the common oversize return, as highlighted in Figure 1.3. From this figure, the following major process flows can also be described.

Total Oversize Return is the combined mass flow of mostly coarse material that has been rejected by the eight fine screens. There is only one oversize return conveyor on which the material is returned to the crushing plant.

Total Fine Screen Feed is the total combined mass flow of feed that moves from the fine ore bins to each of the fine screens for sizing (eight streams). At steady state, this is equal to the tertiary crushing throughput, or fine ore bin feed rate.

Total New Mill Feed is the total mass flow into the milling circuit (four trains). It is equal to total screen underflow and consequently, the *Total Fine Screen Feed* minus the *Total Oversize Return*.

Recirculating Load is the ratio of total feed that is returned as oversize to the crusher for reprocessing, i.e. *Total Oversize Return* $\div$ *Total Fine Screen Feed*

Figure 1.2: Banana screen used for fine ore classification.

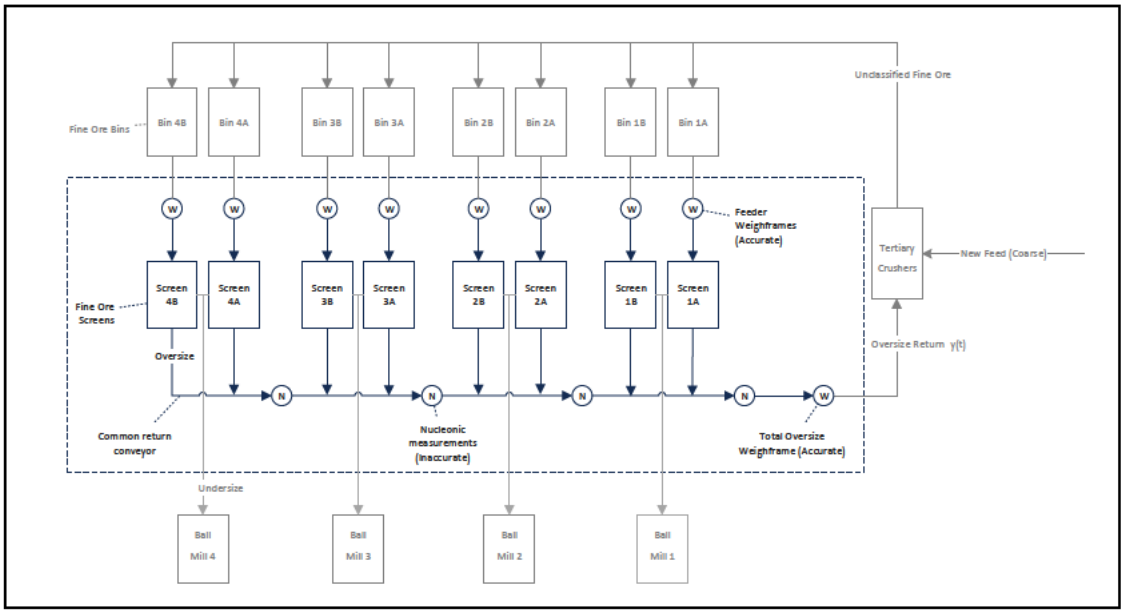

Figure 1.3: The fine screening process being examined.

The fine screening area at Newmont Boddington Gold Mine forms the only connection point between the fines crushing plant and the rest of the concentrator. Subsequently, the performance of this circuit has significant impact on the achievable throughput and ultimately metal production. An increase in recirculating load due to a drop in screening performance not only reduces the available feed to the mill in the short term, but also places additional and unnecessary burden on the tertiary crushers. This reduction in efficiency is thus compounded over the longer term into increased wear on crusher components and increased power usage. As noted by Rotich et al. (2016), poor screen classification is a major contributor to the high energy use associated with poor comminution performance in solid particulate industries.

Figure 1.4: Top deck of banana screen.

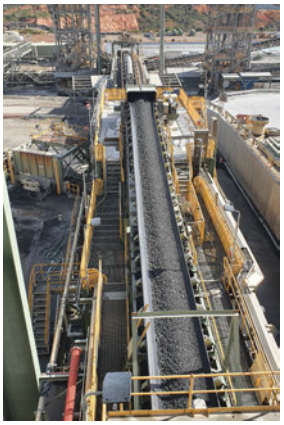

Figure 1.5: Oversize return conveyor.

1.3 Motivation

While the individual fine screen feed rates and Total Oversize Return is known, there is a lack of instrumentation to reliably measure the rate of individual screen oversize. While there exists a measurement of oversize for each pair of fine screens (per mill train), the nucleonic instruments shown in Figure 1.6 are not linear throughout their measurement range and require significant conditioning to be representative of the actual oversize. The compensation scheme currently used assumes that the fraction of oversize is equal across all four milling trains (and all eight screens), leading any increase in oversize either due to measurement error or performance degradation to be distributed evenly across all of the units. An inability to easily attribute which screen is the source of increased oversize makes it difficult to monitor circuit performance and possibly mitigate any issues before throughput is affected.

Figure 1.6: One of four nucleonic weightometers on the oversize return conveyor.

Attempts have been made to install additional instrumentation to overcome these issues. Ideally, the installation of an additional four weightometers on the oversize conveyor so that the oversize on all eight fine screens could be measured would be a simple solution. However, the physical construction of the plant left little room for such an arrangement, making retrofitting impractical. Similarly, it was found that there was not enough room to replace the existing nucleonic devices with a load-cell type weighframe that offers greater accuracy without significant expenditure. Finally, a trial installation of a laser scanner type instrument also proved to be unreliable. While initially promising, the performance was unacceptable much of the time due to interaction with dust and steam in a challenging environment.

Despite the challenges, the existing control arrangement has performed acceptably over time. Historically, milling throughput had been constrained by the maximum crushing output. However, recent upgrades to increase the capacity of major conveyors has facilitated much higher crushing rates, placing increased focus on downstream processes. Due to the high throughputs involved, even a modest improvement to fine screen performance in percentage terms can add substantial economic value.

1.4 Project Aim

Real-time determination of screen efficiency would allow early identification of failure while also informing future maintenance strategies. With a robust, continuous measurement of individual screen oversize, proper balancing of the screen feed to optimize mill throughput could be achieved. There is also potential for the nucleonic measurements to be removed entirely which would reduce the radiation risk on-site as well as reduce ongoing maintenance costs. With these benefits in mind, while statically determining these values offline from historical data has merit, this project aims to ultimately develop an algorithm to perform online parameter estimation in the form of virtual measurement points for continuous use by the DCS.

1.5 Objectives

In facilitating the success of this project, the following objectives have been set:

1. Review existing theory, models and methods for determining fine screen efficiency.
2. Define the structure of the model that best represents the system.
3. Research and compare algorithms used for parameter estimation, in the context of the defined model structure and fine screen efficiency in general.
4. Using the selected algorithm(s), develop an application to identify the parameters of the system model online, as a measure of fine screen efficiency.
5. Verify the performance of the model against actual plant operation.

6. Use the derived model to assess the impact of adjusting the fine screen feed ratio on mill throughput.

If time allows and the above objectives are met, this project will also:

7. Implement an online optimiser in DCS to balance the fine screen feed ratio.
8. Implement an alarming and monitoring scheme for fine screen efficiency.
9. Deploy the model as a portable package.

Chapter 2

Literature Review

This project requires research to be performed across three broad domains. First, a review of the theory regarding the screening process is pivotal, as this will inform the derivation of the plant model. The second area of interest is that of parameter estimation, in which mathematical model structures, potential algorithms and selection techniques will be assessed. The third and final area in the literature that will be reviewed regards the software implementation, from the tools and software that may be of use and the overall approach to the development of an application.

2.1 General Screen Theory

The basic premise of industrial screening is to separate granular particles under a target size from those above it by passing the product through a mesh, grate or some other aperture (Sullivan 2012, p. 1). As discussed by Sullivan (2012, p. 1), whether or not a particle will pass through the given opening is entirely probabilistic and depends on, amongst other things, the difference in size of the opening and the particle itself, and the dimensions of the wire mesh or screening substrate.

There are other several factors that impact screen performance, including (Sullivan 2012, pp. 3-9):

- Size and shape of material

- Feed density
- Moisture content
- Size distribution
- Screening media (material, shape, aperture etc.)
- Condition of screening media (binding/blocking, holed, missing etc.)
- Screen motion (amplitude and frequency of vibration, and bed velocity)

Vibrating screens are one of the most extensively used type of screening in minerals processing for sizing, grading and dewatering applications (Wills & Finch 2015, p. 187). By shaking (vibrating) the screen, the ore bed tends to stratify, allowing coarser particles to move to the top, increasing the probability for smaller particles to fall through (Sullivan 2012, p. 1), and as such providing a significant increase in screening performance. While correct excitation will increase the interaction between particles and screen openings, excessive vibration may be detrimental by making the interactions too erratic and sparse for proper separation (Wills & Finch 2015, p. 185).

For high tonnage situations, capacity and efficiency are equally important. By using a slope that is initially steep but reduces towards the discharge end, banana or multi-slope screens cause fast stratification of particles and hence boast far greater capacity than conventional screens. The softening slope then reduces the average velocity of the remaining material and thus maintaining high efficiency across the entire deck (Wills & Finch 2015, p. 189). The Newmont Boddington Gold Mine utilises banana screens extensively for this purpose, in requiring sustained high throughput and reliable sizing in a challenging environment.

2.2 Screen Efficiency

According to Sullivan (2012, p. 1) a measure of screen efficiency that is used most often is the ratio of the weight of the product that has passed the screen to the total weight of undersize in the feed. Wills & Finch (2015, p. 182) provide a more accurate formula. Given the total mass flow rate of feed to the screen, F, the mass fraction of material finer than the desired size in that feed, f, the mass flow rate of the underflow stream, U, and

the mass fraction of material finer than the desired size in the fine underflow stream, u, the underflow efficiency, E_U can be defined in Equation 2.1.

$$E_U = \frac{Uu}{Ff} \tag{2.1}$$

This definition, assumes that the desired product is in fact the undersize material. Given mass flow in the oversize stream, O, and the fraction of undersize in that stream, o, a measure for oversize efficiency, E_O is then given by Equation 2.2 (Wills & Finch 2015, p. 182).

$$E_O = \frac{O(1-o)}{F(1-f)} \tag{2.2}$$

Wills & Finch (2015) also provide an equation for underflow efficiency in terms of the undersize mass fractions only in the form of Equation (2.3).

$$E_O = \frac{(f-o)u}{(u-o)f} \tag{2.3}$$

It should be noted that while these descriptions of efficiency are the most used, they are only appropriate for measuring performance on the same feed, as the difficulty of separation is not considered (Wills & Finch 2015, p. 182). Similarly, the amount of undesired material present in either the undersize or oversize streams is ignored. Depending on the application, calculating total separation efficiency may be more appropriate (Wills & Finch 2015, p. 183).

Wills & Finch (2015, p. 182) also demonstrate simplifying assumption that can be made to underflow efficiency. Provided that screen panels are intact, it can safely be assumed that the oversize material present in the undersize is negligible, leaving the fraction of undersize (u) to approach 1, giving Equation 2.4.

$$E_U = \frac{U}{Ff} \tag{2.4}$$

Using this simplification, Wills & Finch (2015, p. 183) finally provide an succinct calculation for circulating load in terms of underflow efficiency, C, presented as Equation 2.5.

$$C = \frac{1}{E_u f} - 1 \tag{2.5}$$

Where the circulating load is defined by (Wills & Finch 2015) as the ratio of oversize to undersize flow, $C = O/U$. While this differs from the term *Recirculating Load* defined in Section 1.2, one can appreciate its relevance.

2.3 Parameter Estimation

2.3.1 Linear Regression

Theory on linear models and linear regression is thorough and long-standing. An exhaustive discussion on the developments in the field would be unnecessary but the fundamentals are well discussed in any good statistics books such as De Veaux et al. (2016, pp. 198-262) and Darlington & Hayes (2016, p. 17-85).

Linear regression is the fitting of a linear model to a set of real world data. The line of best fit is normally taken to mean the line that reduces the sum of the squared residuals the most. For a single independent variable, it can be found in the form,

$$\hat{y} = b_0 + b_1 x \tag{2.6}$$

as given by De Veaux et al. (2016, p. 200), where $\hat{y}$ is the predicted value of y. The value of b_1 is the slope of the linear model and can be easily found from the correlation coefficient, r, and standard deviations of both x and y, by Equation 2.7 (De Veaux et al. 2016, p. 201).

$$b_1 = r\frac{s_y}{s_x} \tag{2.7}$$

The intercept, b_0 can then be found by,

$$b_0 = \bar{y} - b_1\bar{x} \tag{2.8}$$

2.3.2 Multiple Linear Regression

Multiple linear regression takes these concepts, and applies them to higher dimensional linear problems. Darlington & Hayes (2016, pp.64) gives the general model linear model of k dimensions.

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_k x_k \tag{2.9}$$

$$= b_0 + \sum_{j=1}^{k} b_j x_j \tag{2.10}$$

De Veaux et al. (2016, p. 837) also provides a set of assumptions and conditions in order for multiple regression to be applicable.

Linearity Scatterplots of y against each regressor should be approximately straight.

Independence The predictor values must be independent from each other

Equal Variance The error variability of each regressor should be similar

Normality The error residuals should approximate a normal distribution

There are several approaches to the estimating the parameters of the multiple regression model. Dobson & Barnett (2018) provides extensive mathematical treatment of both maximum likelihood and Ordinary Least Squares (OLS) techniques. Taken from Rencher & Christensen (2012) and others, for an over-determined system the long-established OLS solution is given by Equation 2.11.

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.11}$$

Where $\mathbf{X}$ and $\mathbf{Y}$ are the predictor and response matrices respectively, for the linear system described by Equation 2.10. Due to the often expensive and likely impossible matrix inversion operation, it is usually more favourable to perform QR decomposition (Shaffer 2020) given by Equation 2.13.

$$\mathbf{X} = \mathbf{QR} \tag{2.12}$$

$$\therefore \hat{\beta} = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{Y} \tag{2.13}$$

Where $\mathbf{Q}$ is an orthoganal matrix and $\mathbf{R}$ an upper triangular one (Shaffer 2020). Such an approach is implemented internally by MATLAB®functions `mldivide` and `fitlm` to solve most non-sparse linear problems (Mathworks 2021*c*, Mathworks 2021*h*).

2.3.3 Polynomial Regression

If a given system model is continuous but exhibits some curvature, linear regression can be extended to include integer powers of the predictor variables (Weisberg 2013)[p. 109]. For a single regressor, the polynomial regression of degree d is given by Equation 2.14((Weisberg 2013)[p. 110]).

$$\hat{y} = b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d \tag{2.14}$$

Quadratic regression is a special case of polynomial regression where $d = 2$ (Weisberg 2013)[p. 110]. For multiple regressors, powers of each variable can be included in the linear sum, as well as their products, yielding Equation 2.15.

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + b_{11} x_1^2 + b_{22} x_2^2 + b_{12} x_1 x_2 \tag{2.15}$$

Provided that each term is either the product of an independent variable and a coefficient, or a constant, it can be considered linear model. In the case of polynomial regression, with some of the predictor variables squared, the model is still 'linear in the parameters' (Frost 2017*a*), and the problem can be solved through normal linear techniques such as Ordinary Least Squares (OLS). This flexibility can invariably lead to a higher number of regressors than required and, as Weisberg (2013) notes, a strategy to select or delete regressors from the model is normally utilised. Similarly, over-fitting can arise if too high of a degree is used as this may force the model to fit the noise as well as the data (Lever et al. 2016). In such cases, some form of cross-validation is recommended.

For single regressors, the MATLAB® function `polyfit` can be used to fit data to an nth order polynomial (Mathworks 2021*l*). Internally, this function simply calls `mldivide` after transforming the single input vector, $[x_1\ x_2\ \cdots\ x_m]$ into Vandemonde form given by:

$$\mathbf{V} = \begin{bmatrix} x_1 & x_1^2 & \cdots & x_1^n \\ x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \ddots & \vdots & \vdots \\ x_m & x_m^2 & \cdots & x_2^n \end{bmatrix} \tag{2.16}$$

By extension, nth order polynomial regression with multiple regressors can be performed using any of the standard linear solver functions by first calculating vectors of each regressor taken to the nth power, and concatenating this into a single matrix.

2.3.4 Quadratic Programming

For systems where constraints or bounds exist on the parameters, OLS, cannot be used. Quadratic Programming (QP) can be utilised however, which finds the vector x that minimises the quadratic function given by Equation 2.17, subject to linear bounds and constraints (Mathworks 2021*f*).

$$\min_x \frac{1}{2} x^T H x + c^T x \tag{2.17}$$

The MATLAB® function `lsqlin` utilises this QP formulation in order to solve linear least squares problems that include bounds or constraints. If no constraints are specified, `lsqlin` simply falls back to using `mldivide` (Mathworks 2021*g*).

2.3.5 Metaheuristic Techniques

Metaheuristic search algorithms can be used to identify the parameters of a system model by generating large amounts of candidate solutions in order to eventually arrive at choice that best minimises the target cost function. The number of specific algorithms are numerous and ever growing (Zelinka & Chen 2018), but leading approaches include Genetic Algorithms (GAs) and Particle Swarm Optimisation (PSO). These algorithms make little

assumptions about the underlying search space and so they are well suited to problems that include non-linear or non-differentiable functions, or incomplete data sets (Fleming & Purshouse 2002). It can be seen in the literature that there are many competing algorithm designs for within the GA and PSO domains.

Genetic Algorithms (GAs) are modelled on the process of natural selection to, over many generations, develop sets of optimal solutions to complex tasks (Lewin 2005). According to Renders et al. (1992), possible applications of the genetic algorithms include on-line, off-line and supervisory optimization and control. An on-line, adaptive controller has been developed by Dangprasert & Avatchanakorn (1996) that employs Genetic Algorithms (GAs) in a simulated environment, though experimental results from Fister et al. (2016) suggested that the time complexity of Genetic Algorithms may be suboptimal when compared with other approaches. Genetic Algorithms (GAs) have also been used for system identification and prove particularly useful in the identification of non-linear systems (Fleming & Purshouse 2002, p. 1224). Chang (2007) notes that classical least squares techniques are only suitable for linear problems and subsequently uses an Real-coded Genetic Algorithm (RGA) to parameterise a model of known, non-linear structure.

PSO is a Swarm Intelligence (SI) algorithm that is designed to mimic the way birds flock to a common goal, through the simple rule-based movements of individual agents through a search space (Zhang et al. 2015). There is some research suggesting that PSO are generally faster to converge on a solution than Genetic Algorithms. Coello & Lechuga (2002) compare the performance of PSO to both sequential quadratic programming and genetic algorithms in respect to the tuning of an Internal Model Controller (IMC) for a greenhouse temperature system. They find that while PSO takes longer than the traditional approach, it provided better performance overall and was faster than the genetic algorithm. Fister et al. (2016) later provided an in depth comparison between many SI algorithms alongside GA and Differential Evolution (DE). The authors find that PSO, was best suited to the task of tuning the Proportional, Integral & Derivative (PID) controller of a 2DOF robot arm mechanism, though it is noted that the low number of generations and population size may favour the early convergence present in PSO.

2.3.6 Artificial Neural Networks

A subset of machine learning, Artificial Neural Networks (ANNs) are modelled on the human brain by approximating the signalling that occurs between neurons to build what is in effect, a complex network of linear regression models (IBM Cloud Education 2020). While there are many types of neural networks, the feed-forward or convolutional type is the most common in the literature. A feed-forward ANN is organised in layers of interconnected nodes that take input data, assign it a weight and if the sum of signals at that node pass a certain threshold, the data is passed to the next layer (Frost 2017*b*). Through the application of a supervised training regime, the predicition accuracy of an ANN is improved over time, allowing them to perform both regression and classication tasks where the structure of the relationship is unknown (IBM Cloud Education 2020). Shanmugam et al. (2021) demonstrate the utility of ANN through the development of a non-linear model for predicting the efficiency of screening coal, from moisture, screen angle and screen frequency data. While their results were mixed, the study demonstrated practical applications of ANN in mineral processing.

2.3.7 Support Vector Machine Regression

Support Vector Machines (SVMs) were originally developed to perform classification tasks, in which it uses kernel methods to map the problem to a high dimensional space and define a hyperplane that best separates data of different classes (Li et al. 2016). SVM has since been adapted to regression problems, in which case it is generally referred to as Support Vector Regression (SVR) in the literature. The standard form of SVR is generally suitable where some acceptable level of error in the prediction is tolerable, as the cost function only considers points on the error margin boundary, and not those beyond it (Sharp 2020). Zhang et al. (2016) train an SVM to predict the sieving efficiency of a screen, through the experimental analysis of the effects of screen aperture, length and inclination. The authors presented a rigourous methodolgy that paired cross-validation with a genetic algorithm to tune the hyperparameters of the SVM. This resulted in sieving efficiency model that was reportedly more accurate than both ANN and empirical formula (Zhang et al. 2016).

2.4 Model Validation

In order to compare two or more models, statistical methods must be employed to estimate the true predictive power of the each model using the available existing data (Cerqueira et al. 2020, p. 1998). For a fair and unbiased estimation of a model's accuracy and generalising ability, it is necessary to assess the models performance on data that it was not trained on (Vanwinckelen & Blockeel 2012, p.1). If there are large amounts of data available, it may be possible to simply split the data set in two, however for sparse or smaller sets of data, a resampling technique may be required in order to maximise the use of the existing samples for both testing and training in a way that does not bias the performance estimate (Brownlee 2018). This resampling is generally known as cross-validation and is often applied to both classification and regression problems (Bergmeir et al. 2018).

2.4.1 k-fold Cross-Validation

Of cross-validation, a k-fold approach is clearly the most discussed in the literature. Pramoditha (2020) presents a simple introduction to this technique, and a visual representation of the procedure is shown in Figure 2.1. Adapted from Pramoditha (2020), the figure also shows the selection of k = 10 folds, which is often presented as the common choice in the literature (Brownlee 2018). The basic premise is to randomly split the data into k blocks, reserve one of the blocks for testing the model, and then use the remaining blocks for training the model. The process is repeated until all of the blocks have been used to test the model. At each testing iteration, a record of the model's error or score is taken and the combined results are usually averaged to form a summary of its performance (Brownlee 2018). As noted by Brownlee (2018), recording the Standard Deviation of the performance indicator may also be beneficial.

Jung & Hu (2015) provides an excellent theoretical treatment on the k-fold cross-validation before presenting an averaging approach to generate an 'ultimate' model from the many candidates that the cross-validation procedure was applied to. Others advocate for repeated cross-validation, though Vanwinckelen & Blockeel (2012) demonstrate that this does not improve the performance estimate once the entire dataset has been exhausted.

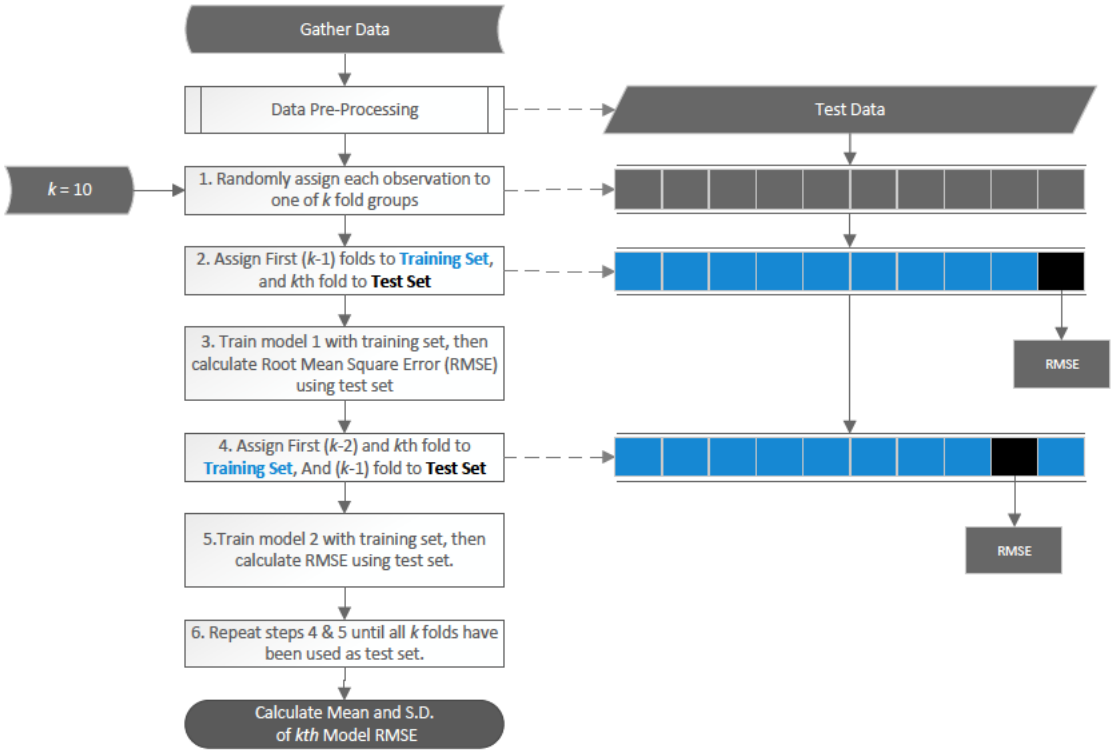

Figure 2.1: A diagram of the k-fold cross-validation procedure (adapted from Pramoditha (2020)).

2.4.2 Other Techniques

There are alternatives to k-fold cross-validation, such as stratified, prequential and Out-Of-Sample (OOS) techniques. Cerqueira et al. (2020) examine many variants of OOS, prequential and cross-validation performance estimation techniques in an attempt to quantify the effectiveness of the different approaches when applied to timeseries data. The authors come to the conclusion for timeseries data with non-stationary models, a repeated holdout OOS technique using Monte Carlo selection of dataset split point to be the best performing technique and that most cross-validation techniques are problematic in this context. Bergmeir et al. (2018), clarifies this somewhat and provides extensive discussion on how k-fold cross-validation is still favourable provided any time dependency is well handled by the model.

2.5 Software and Systems

Implementation of the modelling and parameter estimation techniques discussed could ultimately take place on a variety of systems using many different software packages. This choice however, is confined to the context in which this research takes place, and the specific control system installed at Newmont Boddington Gold. Additionally, a software package is desired that provides relatively simple to use implementations of the required algorithms, and is also able to communicate with the installed control system for the acquisition of data. To this end, MATLAB®was chosen as the ideal option.

An overview of the systems and software relevant to the research follows.

2.5.1 Distributed Control System

A DCS is suite of interconnected software, hardware and computer systems whose primary goal is to provide automated control and operation of an industrial plant (Yokogawa n.d.*b*). As opposed to Programmable Logic Controls (PLCs), which traditionally control a single unit, DCS provides plant-wide control and optimisation, and as such are employed to supervise large or complex processes with large number of control loops (Control Station 2018). Yokogawa's DCS platform, *CentumVP*, installed at Newmont Boddington Gold Mine, provides an extensive combination of Human Machine Interfaces (HMIs), regulatory control, APC, alarm management, a common tag database and many other features within a unified engineering package (Yokogawa n.d.*b*).

2.5.2 OPC Communication

Open Platform Communications (OPC) is a set of standards for the transfer of data between automation and industrial components in a secure and reliable manner (OPC Foundation n.d.). The first specification, Open Platform Communications - Data Access (OPC-DA), was only available to the Windows operating system, and leveraged Microsoft's DCOM framework in a client-server model to provide access to real-time process data (Mahmoud et al. 2015, p. 158). The convenience of OPC-DA's standard interface, is that software can be developed without knowledge of the target device, allowing for the integration of advanced automation solutions in an efficient and const effective manner

(Mahmoud et al. 2015, pp. 155-166).

Newer and more interoperable standards such as Open Platform Communications - Unified Access (OPC-UA) have been developed, though it can be seen across the available literature that OPC-DA remains the de facto standard across multiple industries. Yokogawa's CENTUM Exaopc platform, deployed at Newmont Boddington Gold Mine, uses OPC-DA for data access (Yokogawa n.d.*a*). The MATLAB®OPC Toolbox allows the real-time access of plant data using OPC-DA, enabling MATLAB®programs to read and write data from directly to a DCS (Mathworks 2021*i*).

2.5.3 MATLAB

MATLAB®is a high-level programming language based on matrices that is designed to approximate the natural expression of mathematics. Used in a variety of applications such as signal processing and control systems, the product is extended through the use of Toolboxes, though there is large amounts of functionality found in the in-built functions (Mathworks 2021*n*).

The following subsections, provide a brief overview of some MATLAB®functions that have been found to be relevant to this research:

mldivide

The MATLAB® function `mldivide` or *Matrix Left Division*, is a simple built-in function to solve systems of linear equations of the form $Ax = B$. The actual algorithm employed by MATLAB® to achieve this is dependent on the structure of the input data. According to the documentation, for a non-sparse rectangular array where the number of rows (observations) is much greater than the number of columns (predictors), `mldivide` returns the least squares solution using QR Decomposition (Mathworks 2021*h*).

fitlm

Much like `mldivide`, `fitlm` from the MATLAB® Statistics and Machine Learning Toolbox uses generally QR Decomposition to provide a least squares solution to a system of linear

equations. The main advantage of this function is that it returns a full linear model object that consists of the estimated parameters, residuals and error statistics (Mathworks 2021*c*). While there are a few more options than `mldivide`, such as robust optimisation using Weighted Least Squares, there is no facility to add constraints or bounds.

lsqlin

`lsqlin` from the MATLAB® Optimization Toolbox is a constrained linear least square solver that also takes different code paths dependedent on the type of problem passed to the function. The default algorithm is a quadratic programming iterative method called 'interior-set' (Mathworks 2021*f*), though if there are no constraints given, `lsqlin` refers the solution to `mldivide` (Mathworks 2021*g*). There is also an option to use other algorithms such as 'trust-region-reflective' and 'active-set' though these are more suited to problems with large amounts of data or large numbers of linear constraints, respectively (Mathworks 2021*f*).

particleswarm

The function `particleswarm` is part of the MATLAB® Global Optimization Toolbox, and implements the metaheuristic global search technique, Particle Swarm Optimisation (PSO). As described in Section 2.3.5, this algorithm generates a population of candidate solutions or 'particles', that iteratively move through the search space in order minimise a cost function (Mathworks 2021*j*). `particleswarm` allows upper and lower bounds to be set and has many configuration hyperparmaeters available to be set such as swarm size, inertia range and adaptive neighbourhood size, as well as options for both parallel and vectorized operations (Mathworks 2021*j*).

ga

The function `ga` is part of the MATLAB® Global Optimization Toolbox, and implements the metaheuristic global search technique, Genetic Algorithm (GA). As described in Section 2.3.5, this algorithm generates a population of candidate solutions, that through the application of a fitness function and the mechanisms of mutation and selection, move

through search space to arrive at the global minimum(Mathworks 2021*k*). `ga` allows upper and lower bounds to be set, as well as linear equality and equality constraints. It also has many configuration hyperparameters available to be set such as population size, cross-over fraction and elite count, as well as options for both parallel and vectorized operations (Mathworks 2021*k*).

fitrnet

The `fitrnet` function, from the MATLAB® Statistics and Machine Learning Toolbox can be used to train a feedforward ANN model. The function takes predictor and response data and returns a trained regression neural network object. The model used consists of a single fully connected hidden layer with 10 outputs by default, but is fully configurable, allowing both the number of hidden layers and their size to be set, as well other options for customization, including the activation functions, and initial weights and tolerances (Mathworks 2021*d*).

fitrsvm

The MATLAB® function, `fitrsvm` from the Statistics and Machine Learning Toolbox creates and trains a Support Vector Machine (SVM) regression model based on a set of predictor and response data. According ot MATLAB®, this type of model is only suitable for low to moderate dimensional data (Mathworks 2021*e*). MATLAB® implements what they refer to as epsilon-insensitive SVM regression, that attempts to find an n-dimensional funciton that has as little curvature as possible, while minimising the error between the function and the response variable for all observations (Mathworks 2021*e*). The function allows for k-fold cross validation internally, as well provides a parallell processing option.

cvpartition

The process of partitioning a data set into n random sets for cross-validation is simplified through the use of the `cvpartition` function from the MATLAB® Statistics and Machine Learning Toolbox. This function provides takes as an argument, the number of observations n and returns a cvpartition object that contains the indices of the training and test

sets, for use by other functions such as `crossval`. Multiple types of partitions are able to be generated including leave-one-out, as well as stratified and non-stratified variants of k-fold and holdout cross-validation(Mathworks 2021*b*).

crossval

The `crossval` function is also part of the Statistics and Machine Learning Toolbox in MATLAB® and provides an unbiased estimate of a model's loss (or error) using cross-validation (Mathworks 2021*a*). The function takes arguments for a loss function as well as for predictor and response data. It then uses either the selected partitioning strategy (passed as argument) or the provided `cvpartition` object to issue training and testing data sets to the provided loss function, which should return a loss value such as RMSE for that iteration. The mean of these loss values are then combined for the final estimate of model performance (Mathworks 2021*a*).

2.5.4 Software Development Life Cycle

A life cycle, when prescribed to software development, is a conceptual model that encapsulates how a required software system is to be brought into reality, through its design, implementation, maintenance and eventual decommissioning (IEEE 2017, p. 17), similar to that shown in Figure 2.2. A traditional approach to the development of software follows sequential steps of intitiation, requirement definition, design, construnction and then testing, commonly referred to as the "waterfall" model (IEEE 2017, p. 18). The challenge of this approach is that unless the entire project requirements are intricately known in advance, there is a high likelihood that changes will need to be made at later stages that are not budgeted for, neccessitating a need for a more incremental or iterative approach(IEEE 2017, p. 18).

Alternative approaches have been devised to mediate these issues, such as Iterative Waterfall, Spiral Model and V-Model, though all have their own limitations (Kumar & Rashid 2018). Clearly, the most popular approach in recent literature is a model known as Agile. Ultimately, Agile development breaks a software project up into smaller more deliverable increments, allowing a faster, more flexible programming approach (Atlassian 2021). While most features of agile are specific to managing teams and dealing with clients,

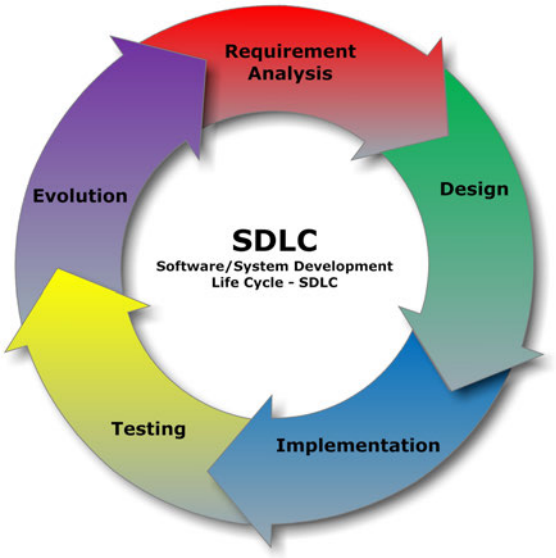

Figure 2.2: The software development cycle (Cliffydcw 2012). Used under a CC Attribution-ShareAlike 3.0 licence (https://creativecommons.org/licenses/by-sa/3.0/)

there is significant merit in following an iterative approach. One specific component of the Agile model that is of particular value is Test Driven Development (TDD), which places testing at the forefront of iterative development (Johnson 2012). A typical TDD methodology would include:

1. Define application requirements based on system architecture,
2. Write acceptance tests,
3. Define unit requirements (modules),
4. Write unit tests for scripts, classes and functions as per Mathworks (2021*m*),
5. Write code to pass unit tests
6. Connect units and test entire application.

This approach to development should not be a linear one, and it is expected that as development and testing progress, both unit requirements and tests would evolve. The critical point is that the all tests should be run when ever a change is made to ensure that functionality has not regressed (Mathworks 2021*m*).

2.6 Research Gap

The prediction of vibrating screen efficiency is an emerging research area and most techniques still rely heavily on laboratory environments or are computationally expensive. It is clear that more work is required here. Despite the numerous other factors involved, measurements of mass flow in the underflow and overflow streams of a real plant give the simplest indication of real performance. While little research could be found that attempts to calculate fine screen underflow efficiency in an online manner, this could be due to the specific nature of the problem in question. Regardless, there is a clear opportunity to apply parameter estimation techniques to solving such a problem.

Approaches for parameter estimation and optimisation are thoroughly presented in the literature. While the evolutionary approaches discussed do seem more suited to non-linear problems, there are many areas where they perform exceptionally well. A further benefit of these techniques is that through the use of an objective cost function, additional constraints and goals can more readily be integrated into the model. On the other hand, the advantages and simplicity of multiple linear regression are well known and should be the technique of choice if the system can be accurately modelled that way. Regardless, much of the research is applied to either simulated or heavily curated environments. The exploration of the competing parameter estimation algorithms in the context of a real world problem is a worthwhile endeavour.

Chapter 3

Methodology

The objectives of this research project are encapsulated by four distinct phases.

Formulation of the System Model

Assessment of algorithms to estimate the parameters of the model

Implementation of the parameter estimator, online optimiser and, alarm and monitoring scheme

Verification of the performance and accuracy of implementations

This chapter will subsequently discuss the formulation of the system model, the selection of an appropriate algorithm in order to meet the stated objectives and the overall architecture that will be required to complete the solution. With this defined, an outline of the verification procedures that will ideally be used to assess the implementation's effectiveness will then be provided. Additional features that complement the final implementation, such as online feed ratio optimisation and an alarm an monitoring scheme will be discussed, and a review of the minimum requirements of each shall be given.

3.1 System Model Formulation

Following on from the review in Section 2.2, there are numerous methods in development to predict the performance of vibrating screens. Most numerical and empirical models of

screens however, depend on measurements that are difficult to obtain on a regular and frequent basis outside of a laboratory setting. Regardless, these models are generally only applicable for the exact physical arrangement and feed on which they were designed. In order for such models to be utilised in this research, a significant amount of experimentation would be required to fully characterise the installed equipment in-situ. Such an approach is not feasible, would result in wholesale disruption and is outside the scope of this project.

In the context of needing to *measure* screen efficiency as opposed to predicting it, the problem is better treated at a macro level through a model based on the mass balance of the entire fine-screening circuit.

3.1.1 Static Model

As starting point in developing a system model, a mass balance of the fine screening circuit at steady state can be made. As shown for a single screen in Figure 3.1, the total feed to the circuit $\dot{M}_{in}$, is equal to the sum of both the undersize and oversize streams from each of the eight fine screens.

$$\dot{M}_{in} = \sum_{n=1}^{8} \dot{M}_{n,\ oversize} + \sum_{n=1}^{8} \dot{M}_{n,\ undersize} \tag{3.1}$$

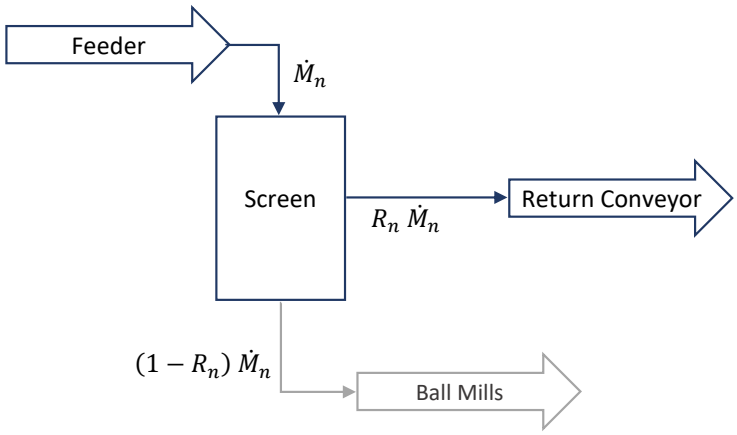

Figure 3.1: Mass Balance of ore streams in and out of fine screen.

Considering just the oversize ore stream from Equation 3.1, it can then be demonstrated that the Total Oversize Return, $\dot{M}_{oversize}$ is equal to the sum of the products of screen feed, $\dot{M}_n$ and some factor R_n.

$$\dot{M}_{oversize} = \sum_{n=1}^{8} R_n \dot{M}_n \tag{3.2}$$

Where,

$$R_n = \frac{\dot{M}_{n,\ oversize}}{\dot{M}_n} \tag{3.3}$$

R_j is the ratio of the mass flow of oversize material to the mass flow of new feed M_n for each screen n, and is synonymous with the recirculating load. As discussed in Section 1.2, at steady state, the recirculating load for the entire fine screen circuit, R_{Total} is,

$$R_{Total} = \frac{\sum_{n=1}^{8} \dot{M}_n}{\dot{M}_{Total}} \tag{3.4}$$

The results derived in Equations 3.2 and 3.3 should be all that is required in order to model the oversize of each screen. It would be beneficial however, to relate this back the general measures of efficiency discussed in Section 2.2. Given equation (2.4) from Wills & Finch (2015, p. 182),

$$E_U = \frac{U}{Ff}$$

and the fact that, $F = O + U$, we can derive a formula that relates the recirculating load value used by Newmont to underflow efficiency in much the same manner as Wills & Finch (2015, p. 183) in equation (2.5).

Defining the recirculating load of any individual screen, n as,

$$R_n = \frac{O_n}{F_n} \tag{3.5}$$

And the mass flow balance at steady state being,

$$O_n = F_n - U_n \tag{3.6}$$

Substituting equation (3.6) into (3.5) gives,

$$R_n = 1 - \frac{U_n}{F_n} \tag{3.7}$$

Rearranging and substituting equation (2.4) into (3.7) then gives,

$$R_n = 1 - E_{U,n} f_n \tag{3.8}$$

Finally, defining underflow efficiency as,

$$E_{U,n} = \frac{1 - R_n}{f_n} \tag{3.9}$$

Arriving at the quite obvious conclusion that the underflow efficiency is directly proportional to $1 - R_n$, the recirculating load or oversize ratio, and inversely proportional to the mass fraction of undersize in the feed to the screen, f_n.

We can also make the reasonable assumption that f_n should generally be constant and equal across all eight screens, provided that the upstream crushing equipment is operating to specification. It follows that any deviation of a single R_n from R_{Total} indicates a change in efficiency of that individual screen n, and that an equal change in all values of R_n (or R_{Total}) would instead suggest a change in upstream performance. The oversize ratio, R_n, as defined in Equation 3.3, is therefore an appropriate single measure of fine screen efficiency for the purposes of this project.

3.1.2 Dynamic Model

Equation (3.2) formulates a static model relating the total mass flow of oversize return as the linear combination of eight inputs. To allow continuous estimation of efficiency, a more dynamic definition is required. The most commonly used approximation for simple processes is the that of a First Order Plus Dead Time (FOPDT) model. By treating the fine screens and return conveyor as a linear, Multiple Input Single Output (MISO), FOPDT system, the individual oversize ratio of each screen can be treated as the unknown input gains and solved given an appropriate data set.

Take the general form of an FOPDT model given in discrete time by King (2016, p. 13),

$$PV_k = a_0 + a_1 PV_{k-1} + b_1 MV_{k-\theta/ts} \tag{3.10}$$

$$a_0 = (1 - e^{-ts/\tau}) bias \qquad a_1 = e^{-ts/\tau} \qquad b_1 = \beta(1 - e^{-ts/\tau}) \tag{3.11}$$

Where β is the process gain, τ is the process lag, θ is the process deadtime, ts is the scan time and k is the current timestep.

A single fine screen stream, n, from feeder through to return oversize conveyor can then be modelled as,

$$y_n(k) = a_0 + a_1 y(k-1) + b_1 x_n(k - \theta/ts) \tag{3.12}$$

Where $y_n(k)$ is the oversize mass flow for a given screen feed x_n at timestep k. Taking the DCS scan interval $ts = 1$ and a bias of zero, this simplifies to

$$y_n(k) = a_1 y(k-1) + b_1 x(k - \theta) \tag{3.13}$$

Finally, as this process only consists of the vibrating screens and conveyors, it is a safe assumption that the process lag is negligible and the only dynamic effect that needs to be considered is pure deadtime in the form of transport delay between each measurement point. Tending τ to zero leads to,

$$\lim_{\tau \to 0} a_1 = 0 \tag{3.14}$$

$$\lim_{\tau \to 0} b_1 = \beta_n \tag{3.15}$$

$$\therefore y_n(t) = \beta_n x(t - \theta) \tag{3.16}$$

Combining Equations (3.16) and (3.2), the dynamic response of the entire MISO system at time t can now be described as,

$$y(t) = \beta_1 x_1(t - \theta_1) + \beta_2 x_2(t - \theta_2) + \cdots + \beta_8 x_8(t - \theta_8) \tag{3.17}$$

$$y(t) = \sum_{n=1}^{8} \beta_n x_n(t - \theta_n) \tag{3.18}$$

or in matrix form,

$$\begin{bmatrix} y_t \\ \vdots \\ y_0 \end{bmatrix} = \begin{bmatrix} x_{1,t-\theta_1} & \cdots & x_{8,t-\theta_8} \\ \vdots & \ddots & \vdots \\ x_{1,-\theta_1} & \cdots & x_{8,-\theta_8} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_8 \end{bmatrix} \tag{3.19}$$

Where the output, $y(t)$, is the mass flow of total oversize return at time t for a given mass flow onto each screen, x_n, while β_n and θ_n are the oversize ratios and total transport delays for screen n, respectively. Given that θ_n are known constants, only β_n needs to be determined for $1 \leq n \leq 8$. Fine screen underflow efficiency can then be found using from equation (3.9), where $R_n = \beta_n$, giving,

$$E_{U,n} = \frac{1 - \beta_n}{f} \tag{3.20}$$

3.1.3 Accounting for Non-Linearities

Through the process of data exploration, it was observed that the initial assumption of linearity was incorrect. As shown in Figure 3.2, this was observed as a non-linearity at higher feedrates in the scattergraph comparing actual total oversize to the oversize predicted by the basic linear model derived in Equation 3.18.

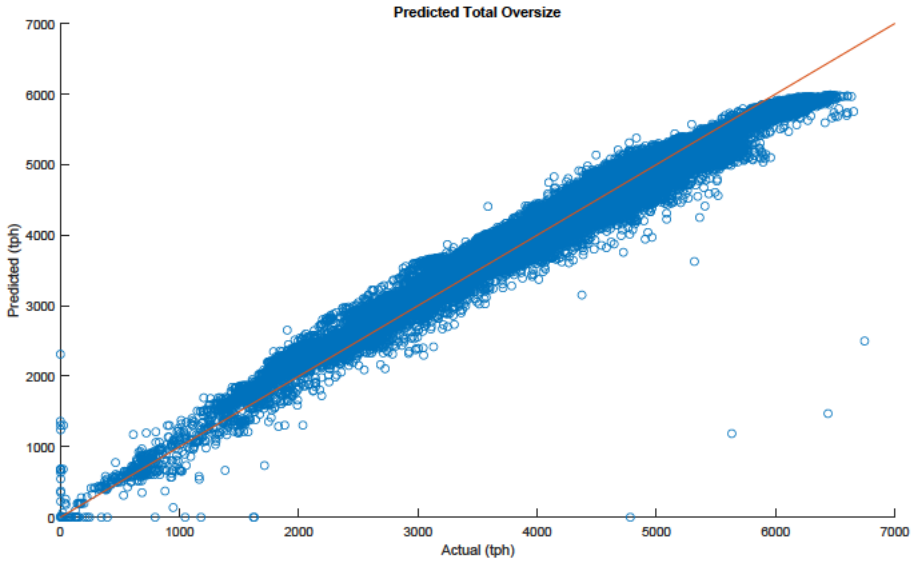

Figure 3.2: Scattergraph of Total Oversize vs Predicted Oversize, linear model exhibiting some non-linearities.

In retrospect, this result is intuitive considering the screening theory presented in Section 2.2. The effect being witnessed is that of increased oversize as the feedrates are

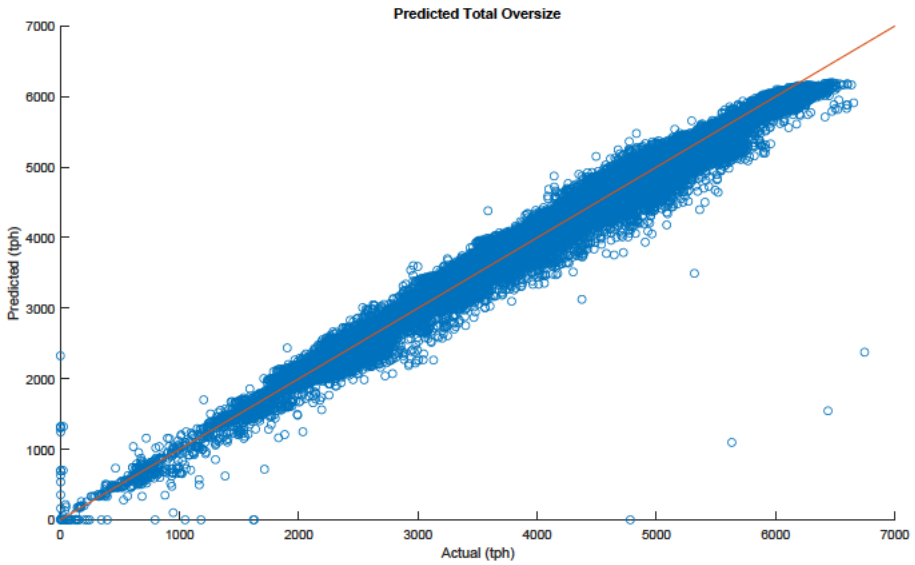

Figure 3.3: Scattergraph of Total Oversize vs Predicted Oversize, quadratic model.

increased past the peak efficiency point for the screen. While the exact structure of relationship is unknown, a quadratic form of the system model can be defined that may better account for these non-linearities. Taking Equation 3.18 and adding a 2nd order term for each regressor, Equation 3.22 can be derived.

$$y_{total} = \alpha_1 x_1 + \beta_1 x_1^2 + \alpha_2 x_2 + \beta_2 x_2^2 + \cdots + \alpha_8 x_8 + \beta_8 x_8^2 \tag{3.21}$$

$$y_{total} = \sum_{n=1}^{8} \alpha_n x_n + \beta_n x_n^2 \tag{3.22}$$

As this quadratic form is still a simple linear sum of terms, the coefficients α and β can still be determined using linear regression techniques, as shown in Section 2.3.3. Performing the regression again on same data-set using the quadratic model in Equation 3.22, yielded the straighter response shown in Figure 3.3. Though some heteroscedasticity is still present and there is a slight tapering off at the upper extremity, the response appears to be a better representation of the system overall, and as such will be used in the final implementation.

In using such a model, the screen oversize ratio R_n becomes dependent on screen feed x_n. From Equation 3.22, the individual oversize of a single screen, y_n is given by:

$$y_n = \alpha_n x_n + \beta_n x_n^2 \tag{3.23}$$

Substituting y_n for O_n in Equation 3.5, the screen oversize ratio R_n as a function of screen feed is then given by Equation 3.25, below.

$$R_n = \frac{\alpha_n x_n + \beta_n x_n^2}{x_n} \tag{3.24}$$

$$R_n = \alpha_n + \beta_n x_n \tag{3.25}$$

3.1.4 Defining Transport Delays

Figure 3.4 provides a visual representation of the delays relating to a pair of fine screens (one mill train). The total transport delay for each fine screen, τ_n, includes the delay from the feeder weighframe, through the screen to the nucleonic weightometer, and from the nucleonic weightometer to the common oversize return weighframe.

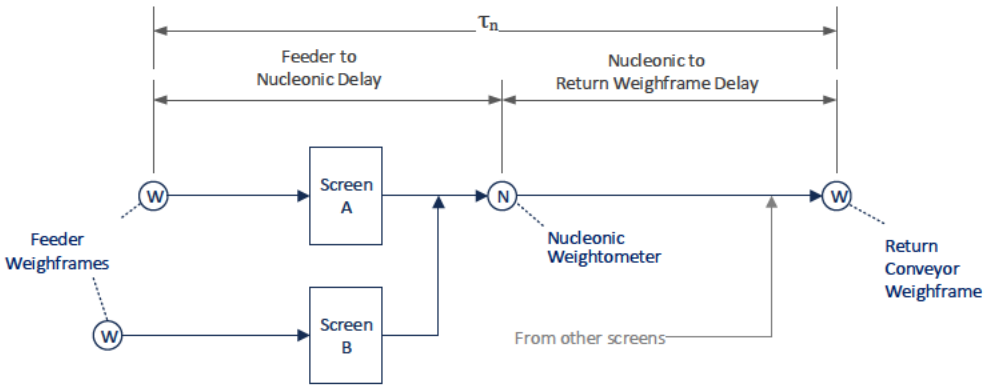

Figure 3.4: Transport delays present from feed measurement to oversize measurement

The values of θ_n are constant and were easily calculated from the dimensions and speed of equipment, and correlated historical data. The transport delays determined for each fine screen are subsequently given in Table 3.1.

Table 3.1: Transport delays to align measurements with total oversize return

n	Stream	Delay (s)		
		Feeder To Nucleonic	Nucleonic to Return Weighframe	Total Delay (θ)
1	Screen 1A	21	15	36
2	Screen 1B	22	15	37
3	Screen 2A	21	21	42
4	Screen 2B	22	21	43
5	Screen 3A	21	26	47
6	Screen 3B	22	26	48
7	Screen 4A	21	32	53
8	Screen 4B	22	32	54

3.2 Assessment of Algorithms

The review of literature in Section 2.5.3 identified several algorithms in the form of MATLAB® functions that may be suited to the task of parameterising the system model. In meeting objective 3, a comparison of the features and performance of each method needs to be performed so that the ideal implementation can be constructed. Firstly, the generic requirements shall be outlined as well as the tools that are available for use. The shortlist of candidate algorithm will then be presented without comment, and an assessment of these algorithms against the requirements will be made.

3.2.1 Requirements

The problem to be solved is that of the estimation of multiple parameters that characterise a linear, MISO system, as defined in Section 3.1. This can be framed either as an identification problem or an optimisation problem with a singular objective, namely to find the vector of $\boldsymbol{\beta}$ values that minimise the difference between predicted oversize, $\hat{y}$ and actual oversize return, y.

For an ideal algorithm in this context, the following characteristics must be present:

Suitable in that the algorithm is capable of solving the given objective function using the defined model structure without unnecessary complexity.

Accurate in terms of both the predicted oversize value and the estimated parameters themselves.

Repeatable so that solution is stable and gives the consistent outputs.

Efficient in order to provide prompt calculations with minimal computing resources, and this more easily re-deployable.

The remainder of this subsection discusses the tools and procedures that will be employed to assess the algorithms against the above measures.

3.2.2 Tools

While the general suitability of an algorithm can be assessed through a review of the literature alone, the remainder of the required characteristics will need to be benchmarked against historical data. This will require the following resources:

1. OSISoft® PI Historian Data Archive
2. Windows PC or Server meeting recommended requirements
3. MathworksMATLAB® R2018a or later
4. MATLAB® Interface for OSISoft®PI System
5. MATLAB® Statistics and Machine Learning Toolbox
6. MATLAB® Global Optimization Toolbox
7. Network connection between MATLAB® host machine and OSISoft® PI server

3.2.3 List of Candidate Algorithms

From the discussion in Section 2.5.3, the shortlist of candidate algorithms are as follows:

mldivide (Matrix Left Division)

fitlm (Fit Linear Model)

lsqlin (Constrained Linear Least Squares)

particleswarm (Particle Swarm Optimisation)

ga (Genetic Algorithm)

fitrnet (Fit Neural Network)

fitsvm (Fit Support Vector Machine)

3.2.4 Suitability Assessment

The first characteristic to be assessed is the algorithm suitability. The assessment contains a list of features that, make the algorithm a realistic and achievable candidate for

Table 3.2: Assessment of Algorithm Suitability

Candidate	Suitability Criteria					Total Score
	Multiple Inputs (**M**[1])	Inter-pretable (**M**)	Bounded	Simplicity	Literature Support	
	(Yes/No)		(1=Bad, 5=Good)			
`mldivide`	Yes	Yes	1	5	5	11
`fitlm`	Yes	Yes	1	4	5	10
`lsqlin`	Yes	Yes	5	4	5	14
`particleswarm`	Yes	Yes	5	2	4	11
`ga`	Yes	Yes	5	2	4	11
`fitrnet`	Yes	No	-	-	-	0
`fitrsvm`	Yes	No	-	-	-	0

[1] **M** denotes a mandatory requirement.

implementation, shown in Table 3.2. Features marked with an **M** denote a mandatory feature that immediately disqualifies the algorithm from further investigation.

The justification for these criterion follows.

Multiple Inputs are required in order to match the inputs used in the derived system model.

Interpretable in that the algorithm is able to specifically estimate the parameters of a system described as per Equations 3.18 and 3.22. A black-box will not provide the transparent structure and interpretability required.

Bounded so that the parameter estimations are kept within the expected range.

Simplicity in order to reduce the development time to suit the relative simplicity of the problem.

Literature Support should be strong so that there is enough resources available to implement the algorithm free from.

Following the assessment, only four algorithms with the highest score were selected for

further testing. This is a critical point that will ensure unnecessarily time-consuming work is not performed. Considering the known, relatively simple model structure, and the mostly black-box nature of the ANN and SVM approaches, these techniques were eliminated from consideration outright. Of the remaining algorithms, `mldivide` and `fitlm` are almost identical, with both using a QR factorisation approach and neither accepting bounds on the parameters. The main difference is the simplicity of mldivide in that it simply returns the estimated parameters while `fitlm` returns a 'linear model' structure. The `lsqlin` however, also provides a simple least squares function but additionally allows bounds and linear constraints to be used.

The two metaheuristic techniques in GA and PSO both accept parameter bounds and constraints, and they are also more flexible in the model structure requirements. This however comes at a greater complexity in setup and large amount of hyperparameters that need to be set. Much of the intricacies of such algorithms are well abstracted away by the MATLAB®functions and may not necessarily be an issue.

With these points in mind and the scores shown in Table 3.2, the candidate algorithms selected for further analysis are:

1. `mldivide` (Effectively QR Decomposition)
2. `lsqlin` (Constrained Linear Least Squares)
3. `particleswarm`(Particle Swarm Optimisation)
4. `ga` (Genetic Algorithm)

3.2.5 Usage and Parameterisation

The following brief subsections outline the basic code required to use and parameterise each of the candidate algorithms that were selected for further analysis in the previous section. Each of these algorithms need to be able to estimate the parameters of both the linear and quadratic model structure. Fortunately, the simple linear sum structures defined in both Equations 3.18 and 3.22 allow for this, as will be demonstrated.

Effectively, all combinations of the model structure and solver algorithm need to solve the following linear overdetermined system with m observations.

$$\mathbf{Ax} = \mathbf{b} \tag{3.26}$$

Where A, is an m x n matrix of predictor variable observations, b is an m x 1 column vector of response variable observations (the actual oversize), and x is an n x 1 vector of unknown coefficients.

For the linear model structure, $n = 8$ and the columns of $\mathbf{A}$ simply form the feed to each screen.

For the quadratic model structure, $n = 16$, and the columns of $\mathbf{A}$ are extended with the the square of each of the screen feeds, as described in Section 2.3.3.

It follows that the overarching requirement of the following algorithm implementations is that they all accept an m x n matrix of predictor observations and an m x 1 vector of response observations, and return n coefficients.

mldivide

As previously stated, for an overdetermined system and non-sparse rectangular matrix, `mldivide` employs a QR decomposition based solver function to solve the linear system $Ax = b$. MATLAB® defines a left-leaning backslash, '\', as the built-in operator to perform `mldivide` operations. It is not possible to set bounds or constraints on the estimated parameters and there are no other configuration options required when calling `mldivide` (Mathworks 2021*h*).

Usage:

```
% For linear system Ax = b
x = A\b;
```

lsqlin

The constrained linear least squares solver, implemented by the `lsqlin` function in MATLAB®, applies different solver method dependent on the arguments provided. Most importantly if constraints or bounds are provided, a QP method will be employed (Mathworks

2021*g*). `lsqlin` takes many optional arguments, though most are not needed for this problem.

Knowing that it is physically impossible for the coefficients to be outside the range of 0 to 1, lower (`lb`) and upper (`ub`) bounds of all coefficients will be set to 0 and 1 respectively, as shown in the below listing. The `lsqlin` function also outputs diagnostic information on when the solution converges so an options structure will also need to be passed to the function to disable this feature. The final consideration is the omission of unneeded arguments in the main function call. This can be achieved by setting irrelevant arguments to an empty array with `[]`, as shown in the following listing.

Usage:

```
1        lb = zeros(1,n);
2        ub = ones(1,n);
3        opts =  optimset('display','off');
4        % For linear system Ax = b
5        x = lsqlin(A, b, [], [], [], [], lb, ub, [], opts);
```

particleswarm

Compared to `mldivide` and `lsqlin`, the usage of the `particleswarm` function in MATLAB®, is more involved as the algorithm makes no assumptions of the model structure. `particleswarm`, requires a function handle to a cost function that can only take one argument, the vector of coefficients, and return the cost of that combination (Mathworks 2021*j*). In order to integrate the training data, a strategy of nested anonymous functions must be used, as shown in the below listing.

Also passed to the `particleswarm` function, is the number of coefficients, n, the lower (`lb`) and upper (`ub`) bounds, and an options structure (`opts`). While there are a plethora of ways to customise the swarm, the default options were generally used, with the exception of the those shown in the following listing.

Usage:

```
1        lb = zeros(1,n);
2        ub = ones(1,n);
3        opts = optimoptions('particleswarm',
```

```
                    'SwarmSize', 100, ...          % default = min(100,10*nvars)
                    'FunctionTolerance', 1e-7, ... % default = 1e-6
                    'UseVectorized', true, ...     % default = false
                    'display', 'off');             % default = 'on'

        % For linear system Ax = b
        % Cost function (Vectorised RMSE of oversize model)
        cost =  @(x, A, b) sqrt(mean((A*x' - b).^2, 1));

        % PSO function can only have one input argument
        trainFunction = @(x) cost(x, A_train, b_train)';

        x = particleswarm(trainFunction, n, lb, ub, opts);
```

ga

The configuration of the Genetic Algorithm (GA) function `ga` in MATLAB®, is similar to that of `particleswarm`. It also requires a function handle to a cost function that can only take one argument, the vector of coefficients, and return the cost of that combination (Mathworks 2021*k*). Once again, the training data is included through the use of nested anonymous functions as shown in the below listing.

The main difference is there is also an option to pass linear inequality and equality constraints in addition to bounds. As this feature will not be used, passing an empty array with `[]` safely ignores these arguments. Also passed to the `ga` function, is the number of coefficients, n, the lower (`lb`) and upper (`ub`) bounds, and an options structure (`opts`). While there are a plethora of ways to customise the genetic algorithm, the default options were generally used, with the exception of the those shown in the following listing.

Usage:

```
        lb = zeros(1,n);
        ub = ones(1,n);
        opts = optimoptions('ga',
                    'PopulationSize', 200, ...     % default = 200
                    'FunctionTolerance', 1e-7, ... % default = 1e-6
                    'UseVectorized', true, ...     % default = false
                    'display', 'none');            % default = 'on'

```

```
9         % For linear system Ax = b
10        % Cost function (Vectorised RMSE of oversize model)
11        cost =  @(x, A, b) sqrt(mean((A*x' - b).^2, 1));
12
13        % GA function can only have one input argument
14        trainFunction = @(x) cost(x, A_train, b_train)';
15
16        mdl = ga(trainFunction , n, [], [], [], [], lb, ub, [], opts);
```

3.2.6 Performance Assessment

The assessment of algorithm suitability in Section 3.2.4 allowed a short-list of four competing algorithm candidates to be selected. Of these, reference implementations were developed in the preceding section to allow for further benchmarking. Each algorithm was to be tested against both the linear and quadratic model structures. The aim of this assessment is not to develop a complete program for online parameter estimation, but to quantify the relative performance of each competing model based on a static set of sample data. In this regard, and to ensure objective fairness in the comparison, the reference implementations adhere to the following:

- Take a pre-processed data set of fine screen feed rates and total oversize observations.
- Return an estimation of the eight (8) unknown values of β_n in the case of a linear model or sixteen (16) unknown values in the case of the quadratic one.
- Use a single process thread at all times, though vectorised operation is allowed.
- Do this only once per execution.

All models will be tested against the same set of data. The data extracted from the PI Historian Data Archive consisted of the following:

- 120000 timestamped observations over a 200 day period.
- Each observation consisted of 8 x Fine Screen feeder mass flow rates, and 1 x Total Oversize Return mass flow rate, all in dry tonnes per hour.
- The period of time covered, is during a period where no serious issues or shutdowns have occurred in the fine screening area.

Given this data set, some pre-processing needed to occur prior to testing. Firstly, all the screen feed-rates were time-aligned with the total oversize measurement as per the transport delays given in Table 3.1. This ensures that data can be treated as 'static' and the simple model structure can be used. With this completed, the final step of pre-processing was to eliminate any observations that contained bad or insufficient data. If a single point in an observation was questionable, the entire row was be discarded. With the sanitisation process complete, the final data set contained a total of 108088 good observations.

With the pre-processed data set and reference implementations developed, a comparison of the algorithm's performance was carried out using the k-fold cross-validation technique discussed in section 2.4, and based on the general ideas outlined by Cerqueira et al. (2020), Vanwinckelen & Blockeel (2012), Pramoditha (2020) and others. As is common in the literature, a standard selection of 10 folds was used (k = 10). At every validation iteration, the RMSE and execution time was recorded for each of the candidate algorithms.

The RMSE summarises the error between the actual y and predicted $\hat{y}$ output over N observations, and is defined as in Equation 3.27, below.

$$RMSE = \sqrt{\frac{\sum_{n=1}^{N}(\hat{y} - y)^2}{N}} \tag{3.27}$$

At the end of the procedure, the mean RMSE, the standard deviation of the RMSE, and the mean execution time provide effective measures of the accuracy, repeatability and efficiency of the algorithm, respectively.

The subsequent cross-validation strategy is as follows:

1. Extract required dataset from PI Data Archive.
2. Pre-process the dataset as per requirements.
3. For each candidate algorithm, perform 10-fold cross-validation as per Figure 2.1.
4. Calculate and record the RMSE and execution time for each candidate at each iteration (10).
5. Calculate the mean of the RMSE results for each candidate algorithm.

6. Calculate the standard deviation of the, RMSE results for each candidate algorithm.
7. Calculate the mean of the execution time results for each candidate algorithm.
8. Tabulate and assess the results, then select the candidate.

Listing D.1 from Appendix D provides the MATLAB® implementation of the cross-validation procedure performed on the candidate algorithms. Partitioning of the data-set was performed using the `cvpartition` function and 10 folds. Each of the reference implementations were wrapped in a cross-validation function that could be passed directly to the `crossval` function and return the required performance measurements. While the model functions themselves were single threaded, parallel processing was enabled to allow multiple folds to be evaluated simultaneously, reducing the overall time taken to perform the validation. The subsequent results from this process is now presented in Table 3.3.

Table 3.3: Algorithm Benchmarking

Candidate	Accuracy[1]	Repeatability[2]	Efficency[3]
Linear Using mldivide	162.42	6.01	0.0148
Linear Using lsqlin	162.42	6.01	0.0089
Linear Using PSO	168.79	21.76	2.4129
Linear Using GA	162.42	6.01	5.2179
Quadratic Using mldivide	138.35	6.47	0.0864
Quadratic Using lsqlin	138.21	6.48	0.0287
Quadratic Using PSO	150.32	11.00	30.4139
Quadratic Using GA	299.84	16.36	114.7158

Cross-validation performed on an Intel i7 with 16GB of RAM.

[1] **Accuracy** is defined as the average of the Root Mean Squared Errors (RMSEs) gathered from the k-fold cross-validation process.

[2] **Repeatability** is defined as the standard deviation of RMSEs results from the k-fold cross-validation process.

[3] **Efficiency** is defined as mean time to convergence on a solution over 10 tests.

3.2.7 Selection

The performance assessment allows for some interesting conclusions. It is clear that for this particular problem, the use of GA and PSO algorithms is simply not warranted. For

both of these algorithms, the time taken to arrive at a solution was orders of magnitude longer than any of the non-stochastic approaches and at the same time provided no benefit in terms of prediction accuracy. This is a somewhat expected result as the system model does appear to be linear in the parameters and is a continuous function. As such, more complicated, non-linear techniques are usually unnecessary.

Although, the accuracy of all of the algorithms when applied to the linear model were comparable, `mldivide` and `lsqlin` were substantially more accurate when used with the quadratic model. Interestingly, the QP approach used by `lsqlin` appears to be considerably faster on all accounts, though `mldivide` still converges in the sub 100 millisecond timerange. Overall, and considering that it allows bounds and linear constraints to be applied to the parameters, `lsqlin` paired with the quadratic model structure appears to objectively be the best approach to solving this problem.

3.3 Software Implementation

With the completion of sections 3.1 and 3.2, Objectives 1, 2 and 3 have been met. A good appreciation of the system model has been developed and an appropriate algorithm has been selected. With these components, an online parameter estimator can be developed to solve the fine screen oversize model. For brevity, this application will be referred to as the Fine Screen Oversize Solver (FSOS) for the remainder of this paper.

The development of the FSOS application has three aspects to consider:

System Architecture that defines the overall structure of the application

Software Development Life Cycle (SDLC) that determines the workflow that will be used to implement the system.

Tools and Methods that will be used in the development.

A discussion of each of these items follows.

3.3.1 System Architecture

In order to meet the requirements of objective 4 and ultimately the aim of this project, an online parameter estimator (FSOS) must be developed based on the algorithm selected in Section 3.2. While the implementation of this algorithm itself is not trivial, it forms but a single component of the entire solution.

The system also requires components that:

- Take user input and interaction
- Accept and store a continuous stream of data from the plant control system
- Apply pre-processing to that data if required
- Determine which parameters can be estimated at any one time (E.g. A screen is offline or the data variability is too low)
- Execute estimation algorithm

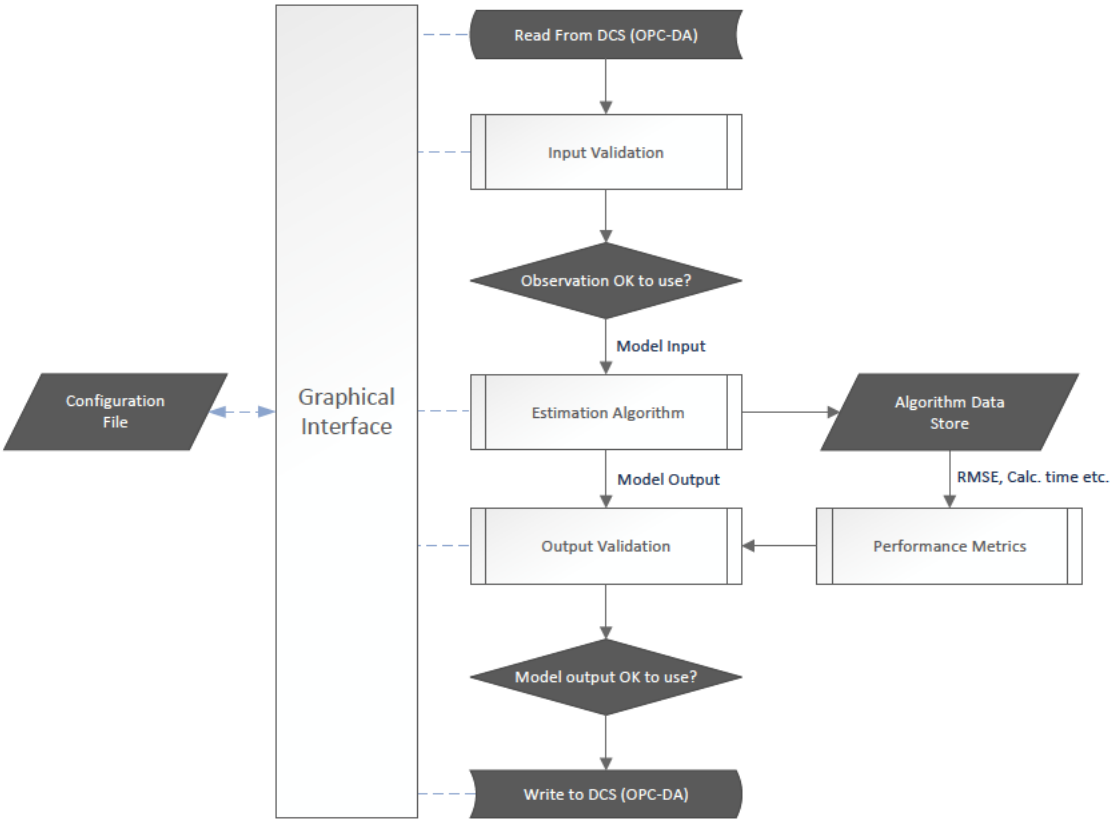

Figure 3.5: Flow chart of the intended system architecture.

- Detect errors in execution and calculate performance metrics
- Apply post-processing to estimated parameters
- Output final parameters to the plant system

The system architecture ultimately defines how these items are to be encapsulated, how they fit together, and required the data flows between each. The basic approach would be to develop each of the above items into stand-alone modules where data must be explicitly passed between each step. Modularity will mostly be be achieved through the use of classes and MATLAB® System Objects where appropriate, and favouring functions over scripts as much as possible. This approach will ensure that as the code complexity increases, the individual components can be refactored or redesigned in isolation.

The overall system architecture is described by Figure 3.5.

3.3.2 Software Development Approach

The development of the FSOS application followed an iterative and test driven approach, similar to that discussed in Section 2.5.4. From the system architecture defined in the previous section, each of the modules were generally developed and tested separately, before connecting each section together and testing the combined operation. Once a working solution was developed, the modules were integrated into an graphical user interface, that will be presented in Chapter 4.

3.3.3 Tools

The tools that will be used to complete the implementation are a superset of those defined in 3.2.2, namely:

1. Windows PC or Server meeting recommended requirements
2. MathworksMATLAB®R2018a or later
3. MATLAB®Interface for OSISoft®PI System
4. MATLAB®OPC Communication Toolbox
5. MATLAB®Simulink
6. MATLAB®Statistics and Machine Learning Toolbox
7. MATLAB®Optimization Toolbox
8. MATLAB®System Identification Toolbox
9. OSISoft®PI Historian Data Archive
10. Yokogawa ExaOPC OPC-DA server connected to plant control system
11. Network connection between MATLAB host machine, OSISoft®PI data archive and OPC-DA
12. Git for Windows, for version control
13. Microsoft Visio, for creation of Unified Modelling Language (UML) diagrams

3.4 Verification

3.4.1 Recording

Following implementation of the FSOS, the systems accuracy and stability will need to be verified. To facilitate this, the following model parameters will recorded in the OSISoft®PI Data Historian at minimal compression and scan rate of 10 seconds:

- 8 x Calculated Fine Screen Efficiency Values (Short term / instantaneous)
- 8 x Calculated Fine Screen Efficiency Values (Long Term)
- 8 x "Calculation Enabled" Boolean Value
- Current Total Oversize RMSE
- Estimator execution time (Last run)

An initial run-in period of two-weeks will allow the system to collect substantial data and give a recorded history to analyse.

3.4.2 Field Verification

Once it is confirmed that system is generally stable and functional, a field verification will be performed. With the assistance of plant metallurgists, the following procedure shall be used:

1. For each fine screen (8), take a metallurgical cut of:
   - Screen feed
   - Screen underflow
   - Screen overflow
2. Record the exact time each sample was taken.
3. Metallurgist to determine % undersize in all three samples for each screen (24)
4. Using received analysis values, calculate underflow efficiency using Equation (2.3) for each screen.

5. Using sampled value of feed undersize fraction, f, verify assumptions were correct and calculate fine screen underflow efficiency for each screen using Equation (3.20) and values for β taken from the historian at recorded the sample timestamp.
6. Compare the results obtained from steps 4 & 5, and report on anomalies.
7. Repeat the above at least twice at temporally distant opportunities, to account for any errors in the sampling process.

It is anticipated that the predicted screen efficiency will not be the same as that obtained from metallurgical sampling, due to the assumptions made around both feed and underflow undersize fractions being constant. However, the predictions should be proportional to the sampling results and represent a good and consistent approximation. Provided that any discrepancies are explained by a constant error term, this may be factored into the model in order to improve it's future accuracy.

3.5 Screen Feed Ratio Optimisation

The final main objective of this reasearch, Objective 6, is to assess the impact of adjusting the fine screen feed ratio on mill throughput. As discussed in Section 1.2 and illustrated in Figure 1.3, each pair of screens feeds a single mill, and there is a facility to control the split of new feed across these screens in terms of the proportion that is sent over Screen A of the pair A and B. There are, of course, constraints on both the maximum and minimum mill feed, and the maximum and minimum feed to each screen. Given that the screen efficiency is a function of feedrate, and that the performance of any two screens may differ, a relatively straightforward parameter optimisation problem is formed, in which the ideal screen feed ratio for a pair of screens at any given total feed-rate might be determined.

3.5.1 DCS Implementation

While the development and analysis of this optimising strategy will be carried out in MATLAB® and presented in Section 4.4, it is anticipated that the process should derive a simple algorithm that can be implemented in the DCS, using standard function blocks. Such a function would take the model coefficients and total requested feed-rate, and output the ratio required to maximise mill throughput.

In addition to the optimisation algorithm itself, the following features will be important:

Modular , per-mill approach. This is necessitated by the fact that a pair of screens feeds a single mill.

Watchdog controlled to ensure that the optimiser only takes action when the health of the estimator is verified

Overridable by plant operators in response to underdesired behaviour or plant disruptions.

Constrained so that the split ratio is only modulated within safe bounds.

3.6 Alarm and Monitoring Scheme

Provided that the FSOS output is stable and sensible, an additional objective of this project is to implement an alarm an monitoring scheme that would inform maintenance and operational personal of a degradation in screen performance. The intention is that this would be implemented entirely in the DCS, with the following features:

1. With a stable fine screen model, record the baseline performance for each screen. This is a simply a measure of expected oversize for a given mean feedrate.
2. Create an HMI alarm for any prolonged instance where a screens calculated oversize exceeds its baseline by some threshold.
3. Create an HMI alarm when any single screen's oversize exceeds the total circuit Recirculating Load by some threshold.
4. Create an SMS Notification or email alert to maintenance personnel based on the above alarms.

For both the alarms and notifications, care must be taken to ensure that they are not generated incessantly. This will be achieved by adherence to AS IEC 62682:2017 (Standards Australia 2017), which provides direction on how alarm systems shall be managed in process control systems. Ensuring, any implemented alarms are useful and not excessive reduces the risk that such an implementation would become noise to the operator.

3.7 Chapter Summary

This section detailed the methodology followed throughout this research in order to meet the objectives established in Section 1.5. Specifically, Objectives 2 and 3 were met in entirety, through both the full development of the system model and subsequent selection of an algorithm to best estimate its parameters. Additionally, outlines of how Objectives 4 through 8 should be achieved was given, in reference to the Software Design, Verification, Screen Feed Ratio Optimisation and Alarm and Monitoring Scheme sections. In completing these objectives, the actual implementation of the FSOS application, the associated testing and further analysis is required, as will be discussed in the remaining chapters.

Chapter 4

Implementation and Results

The previous chapter derived the system model and selected an appropriate algorithm, while defining the requirements and construction of the FSOS application. This chapter provides a detailed analysis of how each of the functional modules defined in Figure 3.5 were ultimately implemented. While the FSOS appication itself was written in MATLAB®, there were some supporting and connecting components that required implementation in the DCS, which will also be discussed. Following the implementation details, the testing and analysis of the FSOS outputs will be presented. As will be shown, the results of this testing necessitated several modifications to the implementation, with mixed results. Completing this chapter, some example outputs from the model implementation are used to help define and illustrate and an optimisation algorithm that could be implemented in the control scheme to improve mill throughput.

4.1 Implementation in MATLAB

This section discusses the modules that were written in MATLAB® in the implementation of the FSOS application. Where appropriate, a discussion of the issues encountered and justification of design choices will be also be provided. Full code listings can be found in Appendix D, as indicated.

4.1.1 Data Input and Output Modules

The acquisition of live plant data subsequent writing of values back to the DCS was achieved with OPC-DA through the use of the MATLAB® OPC Toolbox.

The basic process for initiating OPC-DA communication is as follows:

1. Create OPC-DA server object with `opcda`, passing IP address and ID of the OPC server as arguments.
2. Add an empty group to OPC Server object using `addgroup`
3. Add OPC Items in the form of plant data tags to the group with `additem`
4. Connect to the OPC Server with `connect`.
5. Perform read and write operations as required (`read`, `write`, `readasync`,`readasync`)

The read and write operations listed as step 5, encompass two different methods of OPC-DA communication, that have very different consequences.

Synchronous data transfer can be initiated with the the `read` and `write` function of the OPC Toolbox. These calls are of a blocking type, as the session must wait for the data to be returned from the server.

Asynchronous operations use callbacks to allow processing to continue while the data collection occurs in the background. The `readasync` and `writeasync` functions from the MATLAB® OPC Toolbox can be used, or by setting the OPC group subscription status to 'ON', and reading the `.Value` property of the OPC items within the group.

Communication with the DCS first required the reading of plant data in the form of the instantaneous feed rates to each of the screens, and the total oversize return, both in tonnes per hour. The outputs of the FSOS application, the model coefficients of each screen, would also need to be written to the control system. As the model coefficients only need to be written when the actually change, two OPC groups were created, one for the read operations and one for the write operations.

The OPC-DA groups used and the subsequent DCS tags wherein can be summarised in Table 4.1.

Table 4.1: OPC Tag and Group Requirements

Group	Subscription	Item Tag	Description
Read	1 sec	WY625108_RD.PV	Screen 1A Feed
		WY625128_RD.PV	Screen 1B Feed
		WY625208_RD.PV	Screen 2A Feed
		WY625228_RD.PV	Screen 2B Feed
		WY625308_RD.PV	Screen 3A Feed
		WY625328_RD.PV	Screen 3B Feed
		WY625408_RD.PV	Screen 4A Feed
		WY625428_RD.PV	Screen 4B Feed
		WY625000RECIRC.PV	Total Oversize Return
Write	Off (Async)	WY625108_WR.DT01	Model coeff. α_1
		WY625108_WR.DT01	Model coeff. β_1
		WY625128_WR.DT02	Model coeff. α_2
		WY625128_WR.DT02	Model coeff. β_2
		WY625208_WR.DT01	Model coeff. α_3
		WY625208_WR.DT02	Model coeff. β_3
		WY625228_WR.DT01	Model coeff. α_4
		WY625228_WR.DT02	Model coeff. β_4
		WY625308_WR.DT01	Model coeff. α_5
		WY625308_WR.DT02	Model coeff. β_5
		WY625328_WR.DT01	Model coeff. α_6
		WY625328_WR.DT02	Model coeff. β_6
		WY625408_WR.DT01	Model coeff. α_7
		WY625408_WR.DT02	Model coeff. β_7
		WY625428_WR.DT01	Model coeff. α_8
		WY625428_WR.DT02	Model coeff. β_8

The code required to initiate the OPC connection is trivial using the `opcda` function from the MATLAB® OPC Toolbox, and given directly in Listing 4.1, below.

Listing 4.1: Creating an OPCDA client in MATLAB®

```
% Create the OPCDA object
daobj = opcda(host_address, server_ID, 'Timeout', timeout);
```

As private methods of the main GUI application class, two functions were created to facilitate the creation of the OPC groups. Listings D.5 and D.6 in Appendix D, provide the code for the `createReadGroup` and `createWriteGroup` functions respectively.

Figures 4.1, 4.2 and 4.3 show how the configuration was performed in the FSOS Graphical User Interface (GUI). The parameters shown in these figures, were passed to the above functions, from the main FSOS class.

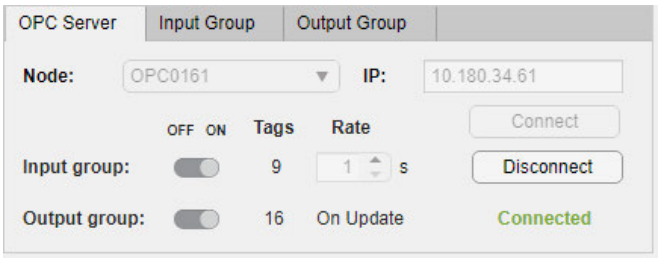

Figure 4.1: OPC client configuration in GUI application

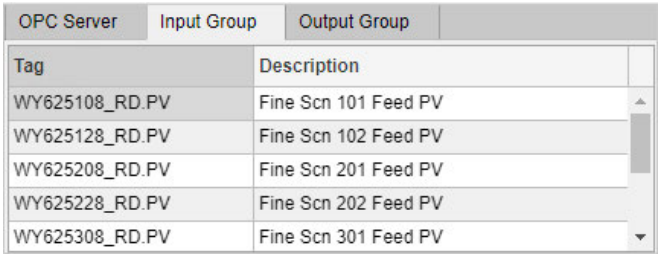

Figure 4.2: OPC read group tags in GUI application

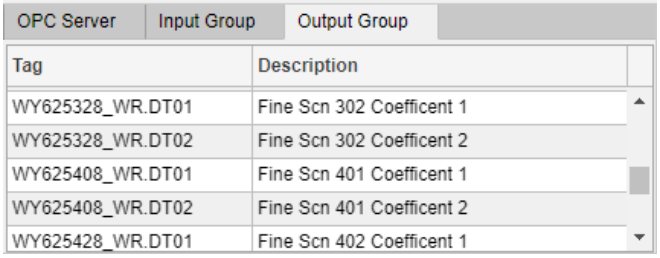

Figure 4.3: OPC write group tags in GUI application

Testing of OPC communication and subsequent input and output functions was straight-

forward and there proved to be no issues encountered that would require further elaboration.

4.1.2 Input Validation

Not all data gathered from the plant in real-time is suitable for being used in the parameter estimation scheme. When performing offline, static modelling, the dataset can easily be sanitised and unsuitable data removed in batches. For a continuously online algorithm however, the input validity needs to be determined in relative real-time.

An input validation module was created to meet this aim, by evaluating the instantaneous inputs with the following criteria.

Low variance across the fine screen feed measurements results in large covariance and model instability. The exclusion of consecutive observations where the standard deviation is low is essential in ensuring ongoing richness of the dataset.

Low throughput , detected as a low overall oversize return measurement is common disturbance that can persist for extended periods, such as shutdowns. Filling the buffer with such observations would be detrimental to the model.

Outliers Due to plant disturbances or instrumentation errors, some observations are clearly well outside of the expected range and should be ignored.

Invalid Values as a result of communication or instrument failure lead to elements of the observation being invalid and as such, the entire observation must be discarded.

Listing D.7 in Appendix D, defines a MATLAB® system object that validates the inputs before they are passed to the solver datastore. A system object is a built-in inheritable class that has features common to discrete signal processing type tasks. This allows the current and past states of the input signal to be saved between execution steps, and used in its internal algorithms before generating the next output.

Taking a vector of the most recent observation data, the `ValidateInput` system object internally calculates a moving average of the last 10 samples, and analyses variance, magnitude, and validity of the data. If one of these criteria falls below the tunable

threshold values, the input it not passed through to the output. Figure 4.4, perhaps better illustrates the approach.

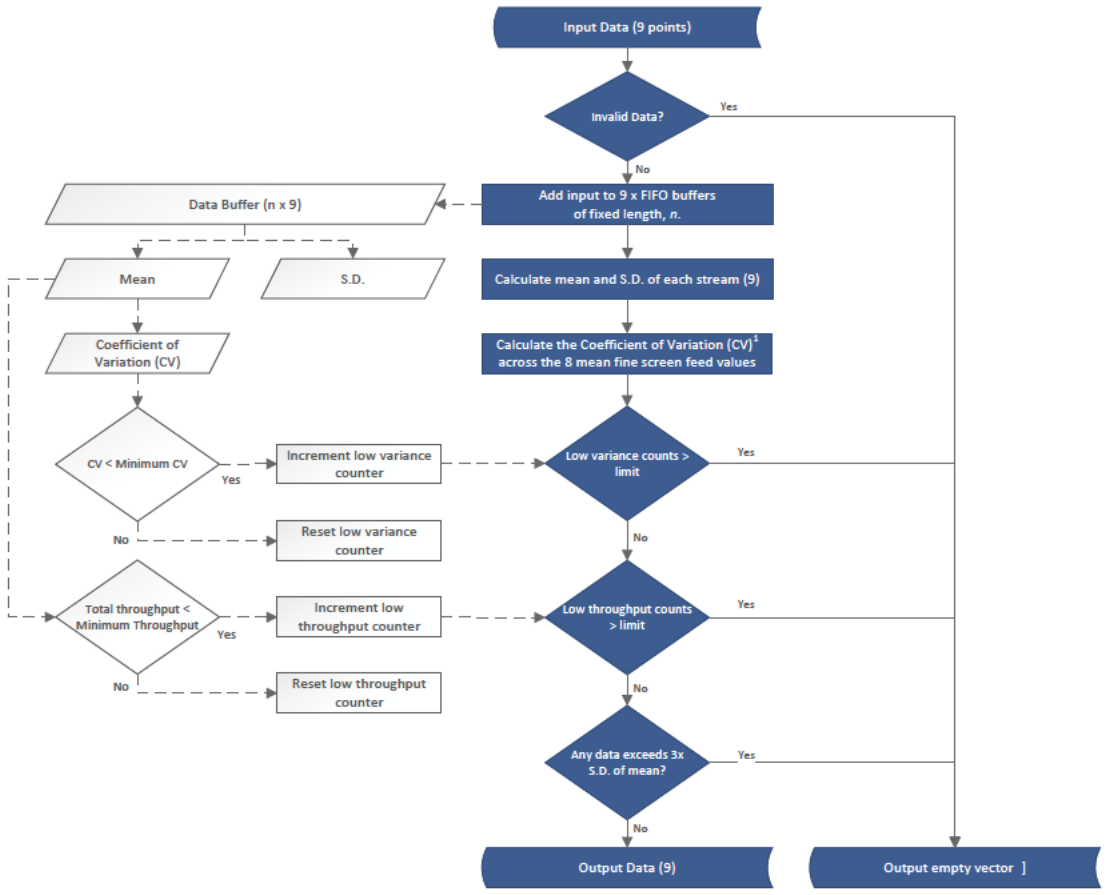

Figure 4.4: Block diagram of ValidateInput class implementation.

[1] **Coefficient of Variation** is defined as the standard deviation divided by the mean, and presents a standardised measure of variance for any feed-rate.

As noted above, the Coefficient of Variation (CV) was used to measure the the variance between the feed to each screen. Initially, the unstandardised standard deviation was used but testing highlighted that the magnitude of this measure was highly dependent on mean feedrate and would dip below the variance threshold when screens were taken offline. An attempt at using the interquartile range was also trialled, but found to suffer similar issues. The coefficient of variation, defined as the standard deviation divided by the mean of the sample, was ultimately found to be a more stable measure of screen feed variance.

While the input validation routine works well, its effectiveness has been found to be fairly dependent on the selection of parameters such as the the minimum variance and minimum throughput thresholds. There is a balance to be found between accepting some poor

observations or inadvertently rejecting some good ones. This challenge will be discussed in later sections.

4.1.3 Model Solver and Predictor Functions

The specific MATLAB® functions used for a single estimation operation were encapsulated in a wrapper function so that interface exposed to the rest of the program was simplified. For the quadratic model structure and selected solver function, `lsqlin`, a pair of functions were created, for the estimation of model parmeters and the prediction of oversize, based on the model.

The function `quadSolver` takes matrices of the predictor (screen feed) and response (total oversize) observations from the data buffer. It expands the predictor matrices by taking the square of each column fitting the quadratic model input data requirements, as discussed in Section 3.2.5. The solver function also accepts upper and lower bounds as parameters, which default to -1 to $+1$, if not supplied. Using `lsqlin` and the given bounds and data, `quadSolver`, outputs the 16 model coefficients, $\alpha_n,\ \beta_n\ [n = 1, 2, \ldots, 8]$, as well as the prediction residuals.

The function `quadPredict` is a complementary function that accepts a vector of screen feed values and a vector of the 16 model coefficients. Using a vectorised version of the the previously defined Equation 3.25, $R_n = \alpha_n + \beta_n x_n$, the predicted oversize ratio for each of the screens is calculated.

The full MATLAB® code for both of these functions is provided in Listings D.11 and D.11, in Appendix D.

4.1.4 Model Solver Class

To maintain a level of modularity while still being able to retain the model state over many iterations, the `Solver` class was created to encapsulate the estimator function defined in the previous section with its running parameters, namely the bounds, and most recent coefficient and residual values. On instantiation of a `Solver` object, the class constructor requires a handle to the solver function, typically quadSolver, though any compatable variant could be used. This flexibility, means that if an improvement on the quadratic

model is found at a later date, it can be replaced with minimal difficulty.

Once setup, the public method `fit`, can be used to retrain the model with a new data set. The implementation is mostly trivial, but the code listing can be found in Appendix D, under Listing D.9.

4.1.5 Model Aggregator

The parameter estimation procedure used in the FSOS application consists both the model itself and the historical data that is used to regress the coefficients. To encapsulate the data and model together, an `Aggregator` System Object class was created, that managed both the storage of data and estimation of model parameters, using the solver class defined previously.

For the data storage, data type class called `CircularBuffer` was also defined, and an CircularBuffer object used as the main data store. The benefit of defining a special data type as opposed to simply using a matrix, was that as the number of observations gets larger, the cost of shuffling every element in the matrix increases. A circular buffer avoids this problem by simply maintaining a pointer to the oldest element in the list, and overwriting that row when new data comes in. While not central to this research, Listing D.12 in Appendix D, provides the complete class definition.

Through testing of the Aggregator class, it was found that some additional flexibility would be desirable. Specifically, the ability to mix the outputs of multiple models might help to create a more stable solution. The use of a separate class to wrap around the solver implementation facilitates this and the instantiation of additional `Solver` objects was trivial. To this end, long and short period models were integrated into the Aggregator, both using data from the shared datastore. With the addition of a *Forgetting Factor* parmeter, q, the two models were mixed by taking the weighted average of the model coefficients from each model, exemplified by Equation 4.1, below.

$$\beta_{\mathbf{out}} = (1 - q) \times \beta_{\mathbf{long}} + q \times \beta_{\mathbf{short}} \tag{4.1}$$

The forgetting factor was able to be modified during operation, through use of the FSOS GUI, as shown in the excerpt given in Figure 4.5.

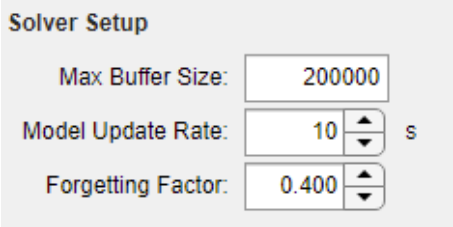

Figure 4.5: FSOS UI - Aggregator/Solver Configuration.

In testing, this addition appears to add some stability, to the solution, though the optimum setting of the forgetting factor and total buffer size remains a challenge. The approach however, is flexible enough that it could be use to mix other models into the solution at a later date.

4.1.6 Additional Graphical Components

The FSOS GUI interface was developed using MATLAB®App Designer. As there is large amounts of boilerplate code generated by this process, it would be superfluous to provide a full code listing though a screenshot of the entire interface is available in Appendix E.There were however, many graphical features developed that add to the visualisation and operation of the FSOS application. An overview of a subset of these components will follow.

Input Plot

Figure 4.6 shows a plot was created to provides a trend of the last 60 seconds of fine screed feed-rates. This plot refreshes on every OPC update to provide assurance that the correct plant data is being received by the model.

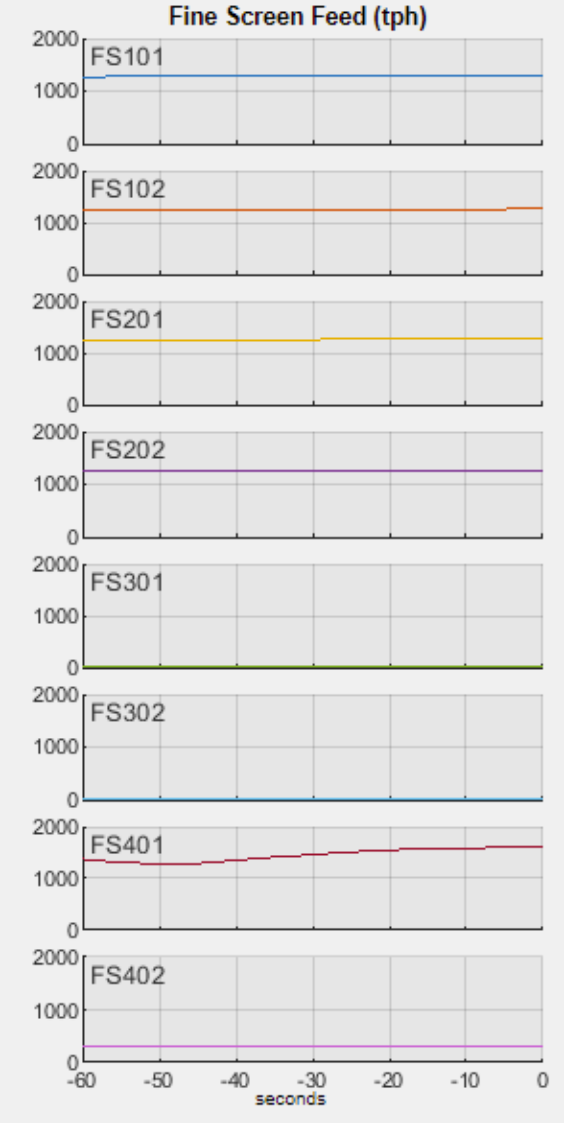

Figure 4.6: Excerpt from FSOS GUI, showing the a timeseries plot of screen feed-rates.

Model Coefficient Table

Figure 4.7 features a coefficient table that displays the current model coefficients, and predicted oversize (in tph) for the given feed. This plot on the right, demonstrates how much feed-rate impacts the oversize ratio for each screen. While the coefficients and associated plot are only updated when the model is updated, the predicted oversize is refreshed on every OPC update.

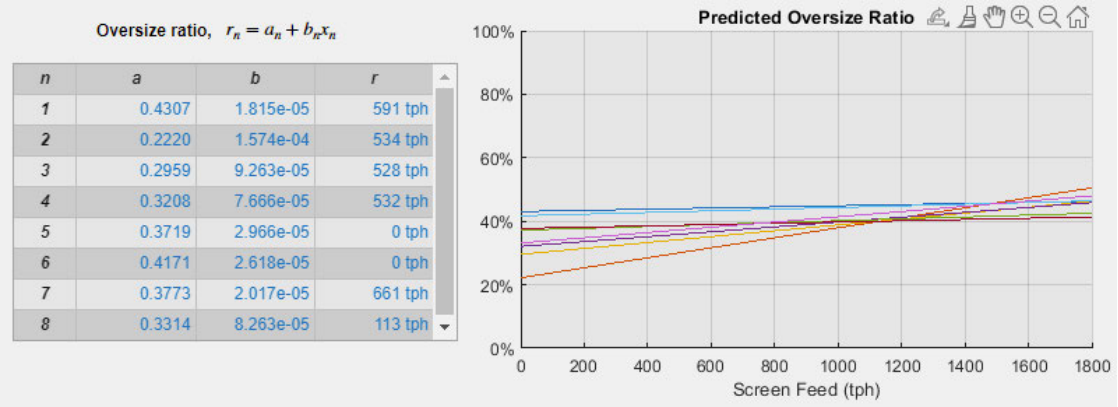

Figure 4.7: Excerpt from FSOS GUI, showing the model coefficient table.

Actual vs Predicted Oversize Plots

Figure 4.8 shows both a timeseries and scatter plot, comparing the actual total oversize, with that predicted by the model.

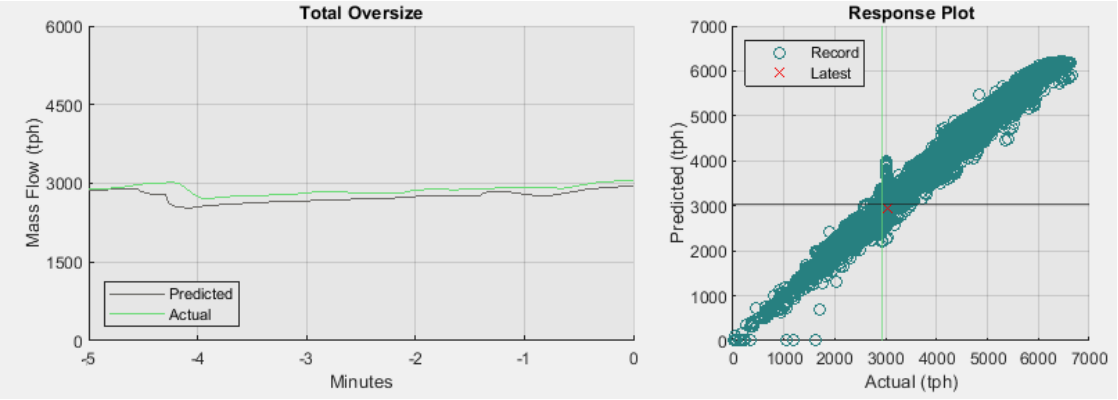

Figure 4.8: Excerpt from FSOS GUI, showing timeseries plot and scatterplot comparing actual with predicted oversize.

4.2 Implementation in DCS

In order for the FSOS application to be of any use, it needs to be connected to the plant Distributed Control System (DCS). As previously stated, this was achieved through OPC-DA communication which necessitates some tags to be created for the reading and writing of data from MATLAB®. While the Yokogawa Exaopc OPC-DA server can theoretically acquire data from any data item in the DCS, the security on the block has to specifically be set to allow write operations. For this reason, dedicated read and write

blocks were created in the DCS to form OPC targets for FSOS application, as specified in Table 4.1 from Section 4.1.1.

4.2.1 OPC Read Blocks

Blocks were created in the DCS to take the weightometer readings and perform small amounts of pre-processing before being read by OPC-DA into MATLAB®. The main requirement was to apply the necessary transport delays as defined in section 3.1.4. The facility to select either the between the raw weightometer reading, feeder speed model, or an average of the two was provided. While the raw reading would ideally be used, this option was implemented in the case that there were issues with individual instruments. The implementation of these features within the native DCS was far simpler than providing them within the MATLAB®program and will make it easier to modify for other engineers in the future without necessitating access to the modelling server.

The DCS code was implemented in function blocks native to the Yokogawa system, similar to that shown in the block diagram given in Figure 4.9, below. The delays in the delay block were set as per that given by Table 3.1 in Section 3.1.4.

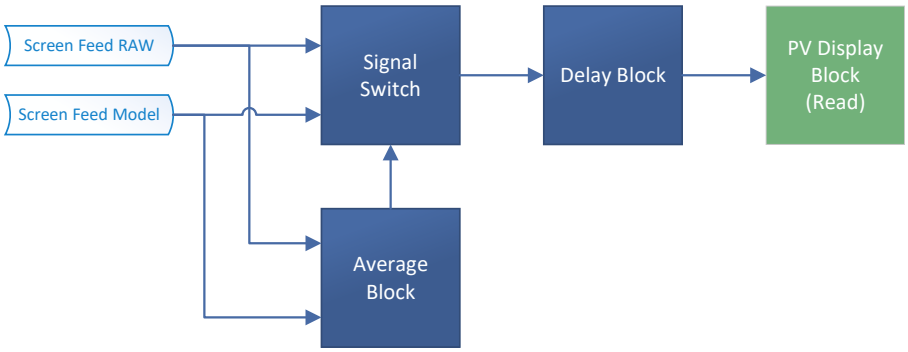

Figure 4.9: Functional diagram of DCS implemented input processing in preparation for OPC-DA Read operations.

4.2.2 OPC Write Blocks

A 'batch data block' for each of the fine screens was created to be the target for OPC to write the estimated model coefficients from the FSOS application running on the modelling server. The batch data block in the Yokogawa DCS simply hold up to 16

numerical data points and offer no other functionality. As such, there was no additional code required, though the fine screen model described in the following subsection, is obtains its model coefficients from it.

4.2.3 Fine Screen Model

Calculation blocks in the DCS allow for short calulation scripts to be executed. The feature storage for up to 8 parameters, take upto 8 inputs and 4 outputs and execute once per DCS scan. This makes them ideal to implement small models and algorithms, and as such are used extensively for this purpose through the plant.

By implementing the system model defined in Section 3.1, and applying the solved model coefficients recieved from OPC-DA to that model, a continual calcualtion of fine screen oversize can occur, regardless of the state of FSOS server. The calculation simply uses the last known model coefficients until a new estimation is received from the server. An additional improvement that can be made at this point, is forcing the model outputs to agree with the actual total oversize. By introducing an error correction gain term, based on the current difference between the predicted and actual total oversize, the oversize ratio of each screen can be gently modified until the mass balance is achieved.

The proprietary syntax used in these blocks is fairly straight forward, though for clarity, just the pseudocode will be provided.

```
Set time constant Tc to large number, eg. 25000
Read the two coefficients (a & b) from batch data block
Read the current fine screen feed (X)
Read the current error (E) between actual and predicted total oversize
Limit coefficients a & b to safe range
Calculate Raw Oversize Ratio (R0) = a + bX
if X > Low feed limit then
    Trim Factor (F) = F + E/Tc
end if
Limit F to safe range
Corrected Oversize Ratio (Rc) = F*R0
Predicted Oversize Tonnes (M) = Rc*F
```

4.2.4 Total Oversize Prediction Error

The final implementation detail in the DCS was the calculation of the model error between the actual and predicted oversize. This used as the error term for the DCS oversize model calculation blocks, in order to correct the mass balance of the circuit. The overall approach is given in Figure 4.10, below. The delays in the delay blocks were again set as per that given by Table 3.1 in Section 3.1.4, in order to align the predictions up with the total oversize return conveyor.

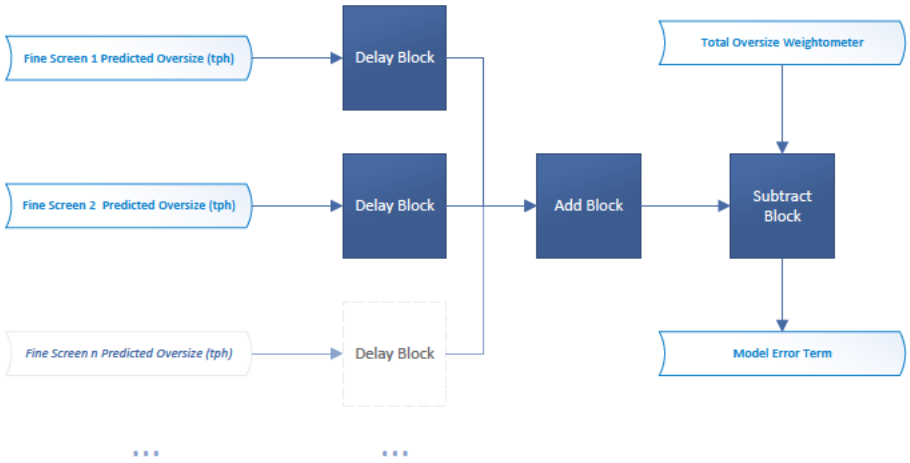

Figure 4.10: Functional diagram of DCS implemented calculation of model predcition error.

4.3 Testing

To verify the reliability of the FSOS implementation and the real-world accuracy of the model outputs, it was proposed in Section 3.4 to perform metallurgical sampling in the field to determine that actual screen efficiency. Unfortunately, due to time constraints this was unable to be achieved at the time of preparing this dissertation. It is anticipated that this still needs to occur in due course, albeit beyond the conclusion of this project.

Despite this challenge, several activities can still occur to assess the validity of the model and overall implementation.

1. Using PI historian, monitor the DCS oversize model to observe well it predicts the total oversize and how well it responds to changing feed conditions.

2. Sensitivity study, using historical data to reevaluate the model using different solver

parameters.

3. Visual inspection of Screens for a subjective assessment of comparative performance

4.3.1 DCS Model Performance

As discussed in Section 4.2.3, an implementation of the finescreen oversize model was built in the DCS, using coefficient updates from the FSOS application. Additionally, this model contains a correction gain, that pushes the oversize ratio on the screens screens closer to actual recirculating load, effectively ensuring that the mass balance is correct.

While the error between the actual total oversize return and the predicted total oversize return is not necessarily indicative of individual screen performance, it still provides a useful metric to the accuracy of the model overall. In particular, if the model coefficients are accurate, the predicted total oversize should still track the actual oversize closely, immediately following a disturbance such a screen going offline, or an abrupt change to feed rate.

Figure 4.11, is a screenshot of trend data from the OSISoft PI Historian. The trend shows plots for the modelled, and corrected, oversize ratio for all eight fine screens, alongside the actual oversize return (WY625000RECIRC) and predicted oversize return (WY625018E).

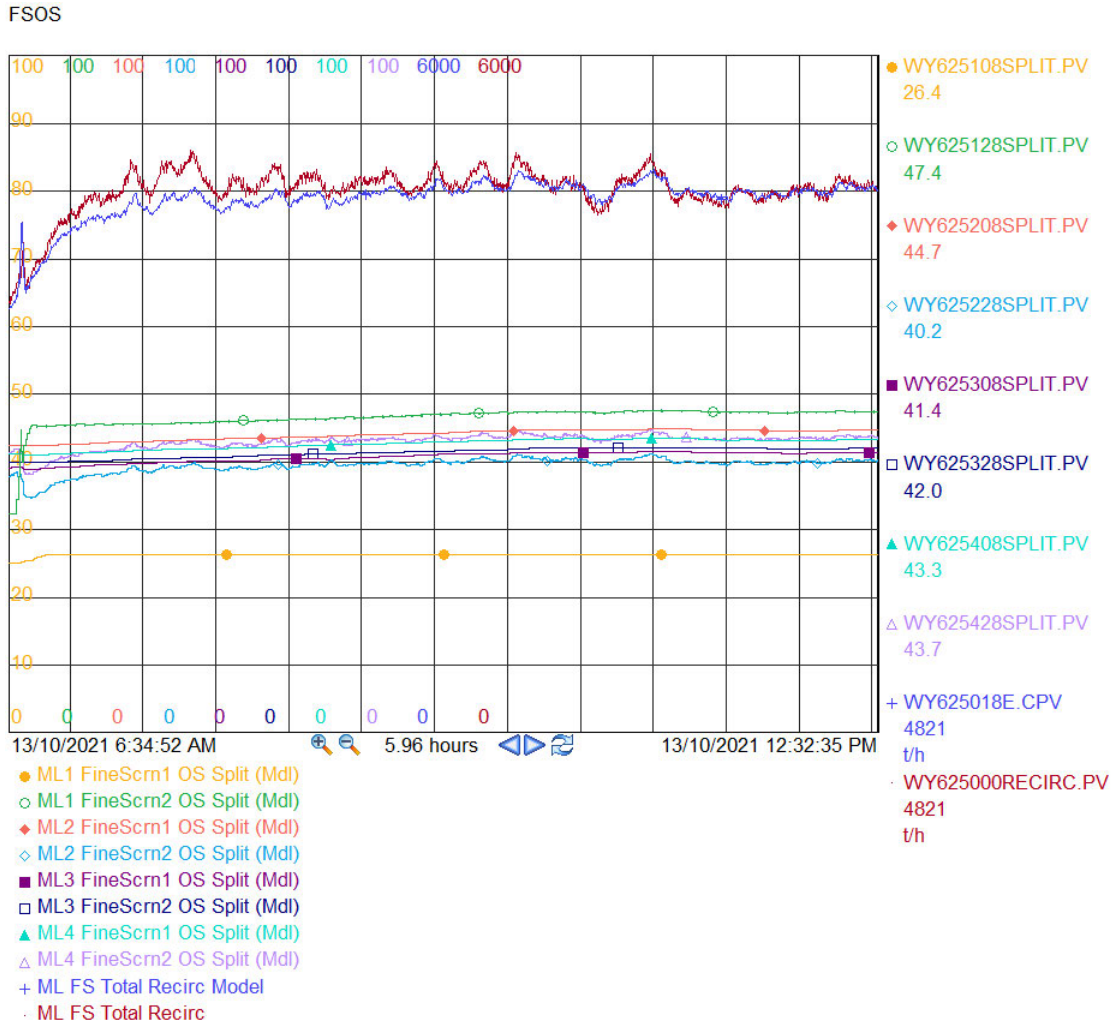

Figure 4.11: Trend of predicted oversize ratio from DCS model following a plant disturbance.

As can be observed, there is initially disturbance to far left of the plot that coincides with a screen 1A being taken down for maintenance. As the feed to screen 1B ramps up to account for the loss of its partner, the predicted oversize return is clearly undershooting. This appears to indicate that the coefficients for screen 1B may not be quite correct. Encouragingly, the model does appear to slowly correct itself over the next few hours, as the error correction term comes into play.

A review of several such events, not pictured, suggests mixed results. Some of the screens appear to be well modelled, while others are inaccurate and certain feed-rates. This appears to occur on screen pairs that have shared extended periods of stability and as such, equal feed-rates. It is assumed that this manifests as severe correlation between the two feedrates in the FSOS model, making the regression unstable. The sensitivity study to follow hopes to address these issues, as will some proposed future work and the concluding

chapter. Besides this, the DCS model implementation appears to be functioning correctly, and the correction term is proven to be of value in stabilising the model outputs, and a good addition to the overall scheme.

4.3.2 Hyperparameter Sensitivity Simulation

Using a large data set observations extracted from the PI Historian that covered a 180 day period at 5s intervals, the simulation script given in Listing D.13 was constructed to emulate the same functions as the FSOS online application. This allowed the entire period to be pushed through the solver, one sample at a time, but at a far greater pace than what can be achieved in real time.

With this approach, multiple passes of the data can be made using different solver configurations for each. The significant hyperparameters in regards to the FSOS model are the window size, and the forgetting factor. For this experiment, fixing the forgetting factor and modifying the window size will demonstrate how the window size in particular effects both the stability of the coefficients and the RMSE of the total oversize prediction.

By setting the forgetting factor to zero, only the long term model will be used. Simulations were then performed with progressively larger windows sizes of 100 000, 250 000, 500 000, 1 000 000 and 1 500 000 observations.

Following the simulation, a moving RMSE of 100000 samples, was calculated for each of the window sizes, with the results in Figure 4.12, below.

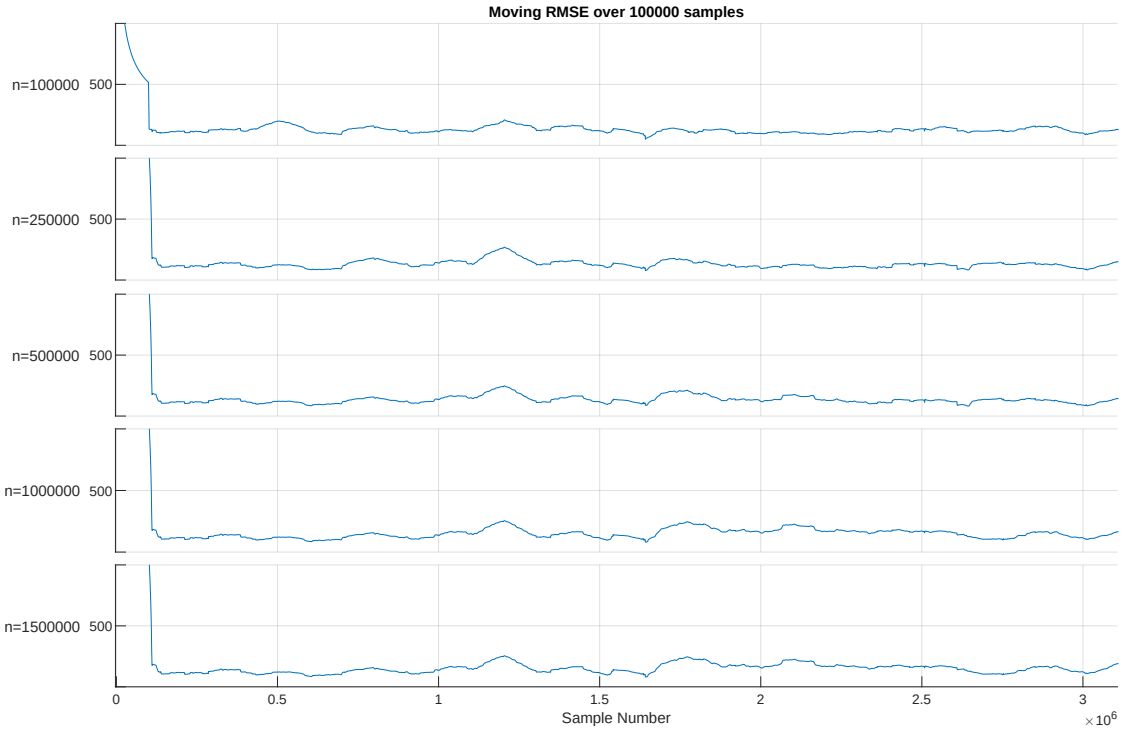

Figure 4.12: Moving RMSE of the five simulation runs of different window sizes (Forgetting Factor fixed at zero)

The results of this analysis are somewhat inconclusive, though the it appears that window sizes less than 500000 samples result in lower RMSE values. Some reasoning behind this is that as screen performance changes overtime, resulting in longer window periods being less adaptive to recent changes.

The coefficients were also captured, during this process. Figure 4.13 below shows the trend of coefficient estimates over the duration of the simulation using a window size of 100 000 samples.

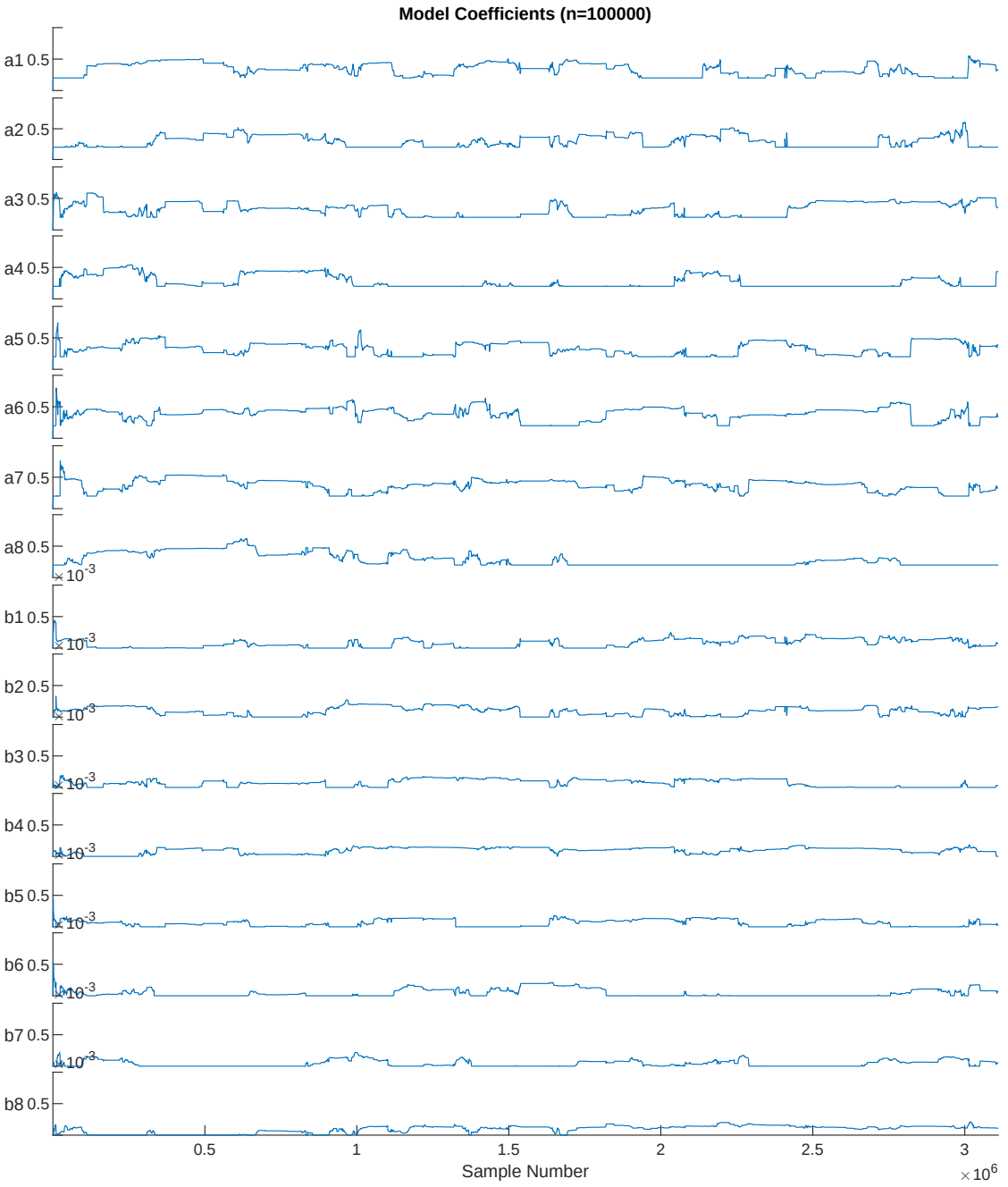

Figure 4.13: Trend of model coefficient estimates with window size, n = 100 000 (Forgetting Factor fixed at zero).

For brevity, the same results for the 1.5 million sample window size is shown in Figure 4.14.

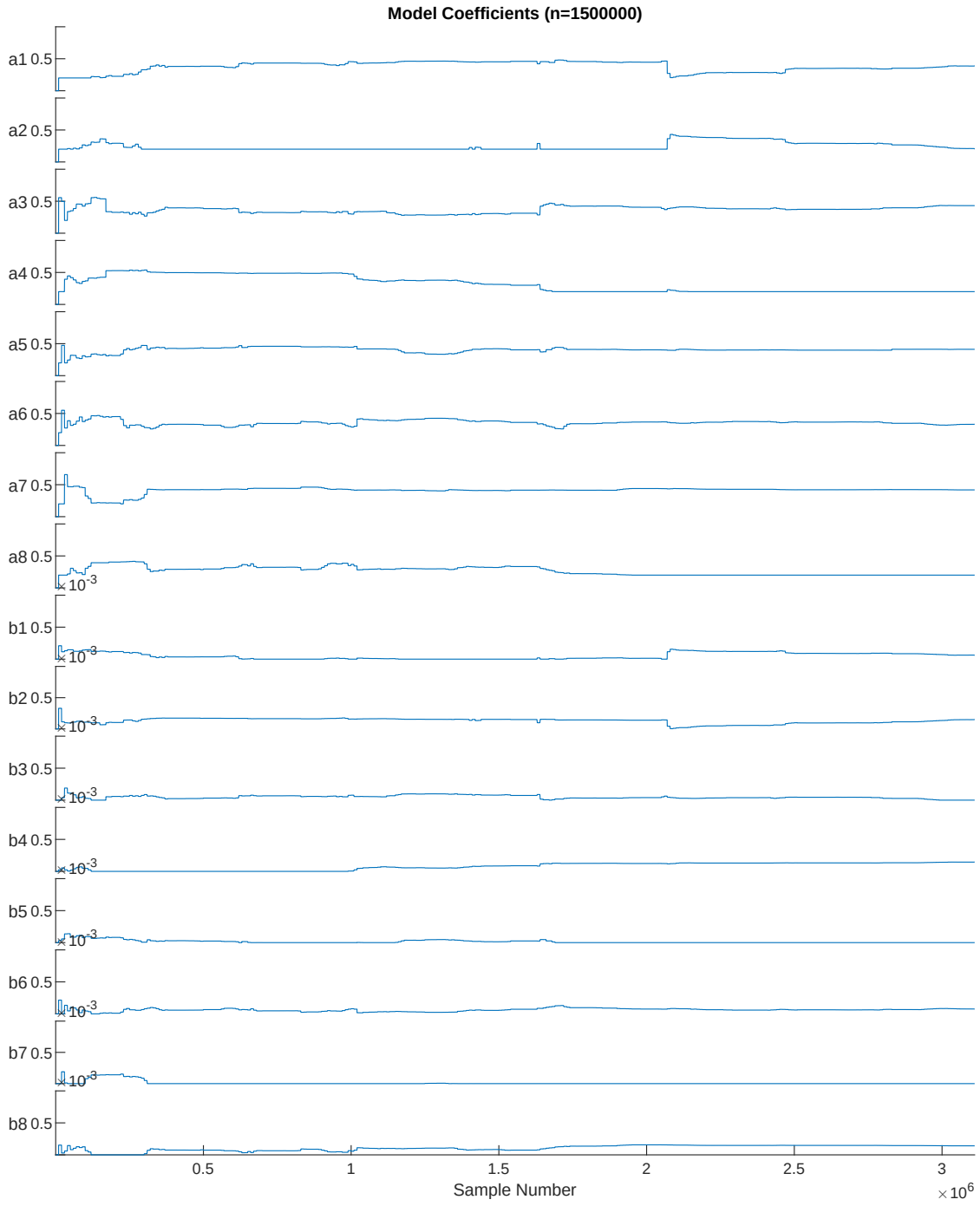

Figure 4.14: Trend of model coefficient estimates with window size n, = 1 500 000 (Forgetting Factor fixed at zero).

Comparison of the previous two figures suggests that the intuition that a larger window size dampens the coefficient response appears to be correct. Larger window sizes result in much less variation in the coefficients over time. Conversely, this results in an increase in model error as the coefficients are less likely to adapt to a changing environment.

4.3.3 Visual Inspection

A visual inspection of the screens was carried out, with a particular interest in the screens that have a high predicted oversize ratio. The hope was that it would be relatively simple to identify which screens had more oversize material than their counterparts.

One screen that appears to consistently have higher predicted oversize than the other was Screen 2A. Footage was taken of the screen, as shown in Figure 4.15, and while it did seem to have high oversize it was difficult to definitively say that it was significantly larger than the other screens. A more detailed review would be required.

Figure 4.15: Fine screen 201 Feed presentation

4.3.4 Discussion

Through the implementation and testing activities several challenges have been identified and there are clear opportunities for improvement. The main concerns are in respect to the stability of the model output, specifically at smaller solver window sizes. While increasing the window size clearly increases the stability of the model coefficients, it has the side-effect of reducing the ability of the model to respond to changes in fine screen performance in the short-term. Given that real-time measurement of fine screen efficiency forms the cornerstone of this project, such a compromise is clearly undesirable.

A more robust approach to fine tuning the model hyperparmeters is likely required, as comparing the moving RMSE of each model in simulation proved inconclusive. Some form time-series appropriate cross-validation strategy may be more appropriate. Aside

from the solver hyperparameters, the test results suggest that input validation could be improved. As the quality of the model outputs depend on the data available for regression, a more nuanced statistical analysis of the inputs at the validation step is warranted. This would result in a larger quantity of informative data in the mode, allowing a reduction in window size.

4.4 Screen Feed Ratio Optimisation

4.4.1 Overview

Based on the fine screen oversize model developed in the previous sections, the effects of increasing the feed rate on the screen efficiency and subsequent mill feed rates can be analysed, as first discussed in Section 3.5. An algebraic analysis of a single pair of screens will first show how the individual feed rates effect mill feed, through increased oversize. This will then be extended to look at the problem from the perspective of the feed split ratio, an existing, manually set parameter in the control system that defines how the feed is shared between the two screens in a pair. A formula as a function of feed rate and screen model coefficients will be ultimately derived.

4.4.2 Analysis

Let z be the mass flow of total feed to a single mill, comprising of the undersize material from two finescreens, A and B.

The total screen feed (oversize + undersize), M_t is split across screens A and B, at rates that shall be denoted by x and y respectively.

$$M_t = x + y \tag{4.2}$$

And the mill feed, z, is thus given by the total screen oversize subracted from the total feed as shown in Equation 4.3, below.

$$z = M_t - M_o \tag{4.3}$$

Given that $M_o = x_{oversize} + y_{oversize}$ and from Equation 3.22,

$$\begin{aligned} z &= x + y - (\alpha_1 x + \beta_1 x^2 + \alpha_2 y + \beta_2 x^2) \\ &= x - \alpha_1 x - b_1 x^2 + y - \alpha_2 y - b_2 y^2 \end{aligned} \tag{4.4}$$

Using the modelled results for Screens 1A and 1B, from the solver developed in previous sections, a test case can be crafted from the recently calculated coefficients.

$$\begin{aligned} \alpha_1 &= 0.430, & \beta_1 &= 1.815 \times 10^{-5} \\ \alpha_2 &= 0.222, & \beta_2 &= 1.574 \times 10^{-4} \end{aligned}$$

A MATLAB®script given in Listing D.14 was used to plot the function using these coefficients across the range of possible feedrates for each screen. As can be seen in Figure 4.16, the result is a smooth concave surface in three dimensions, best described as a partial elliptic paraboloid.

The maximum mill feed, z can then be found at the global maximum where the partial derivatives with respect to x and y are both zero.

$$\frac{\partial z}{\partial x} = 1 - \alpha_1 - 2\beta_1 x \tag{4.5}$$

$$0 = 1 - \alpha_1 - 2\beta_1 x$$

$$x = \frac{1 - \alpha_1}{2\beta_1} \tag{4.6}$$

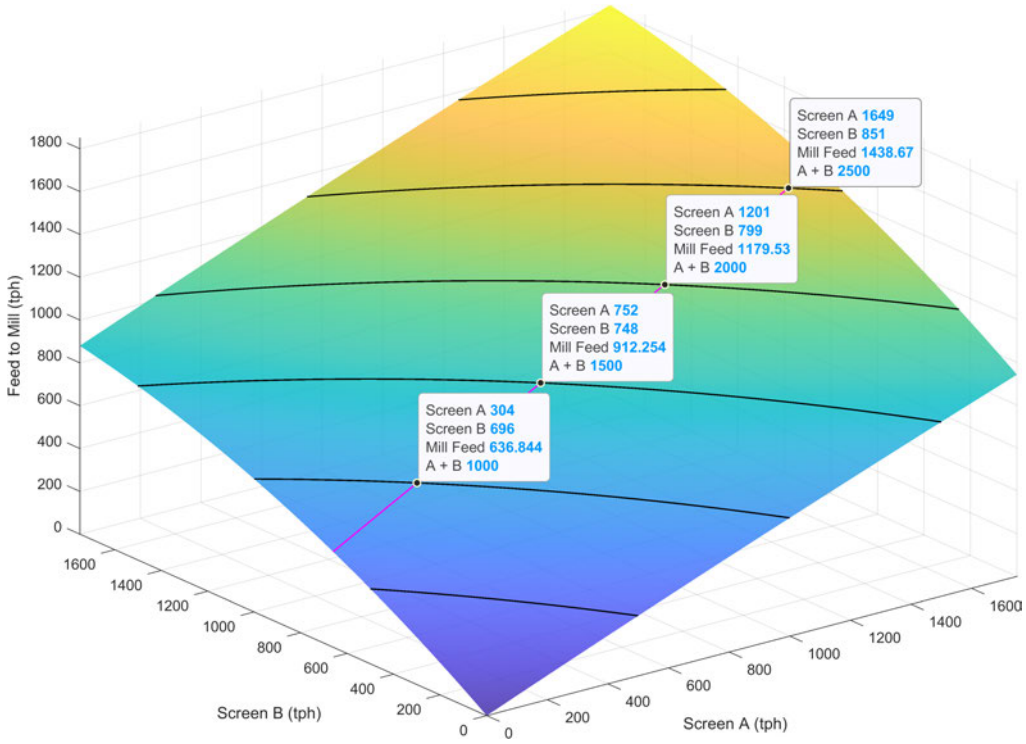

Figure 4.16: Effect of increasing screen feed on mill feed for a pair of screens. Black contours are lines of constant total feed.

$$\frac{\partial z}{\partial y} = 1 - \alpha_2 - 2\beta_2 y \tag{4.7}$$

$$0 = 1 - \alpha_2 - 2\beta_2$$

$$y = \frac{1 - \alpha_2}{2\beta_2} \tag{4.8}$$

A line that traverses the maximum possible mill feed as the total screen feed increases, must follow a path where $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}$ and assume the form:

$$y = mx + c \tag{4.9}$$

Where the slope m,

$$m = \frac{y_{max} - y_0}{x_{max} - 0} \tag{4.10}$$

For the y-intercept term c, letting $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial x}$ and $x = 0$ gives,

$$1 - \alpha_2 - 2\beta_2 y_0 = 1 - \alpha_1 - 2\beta_1(0)$$

$$\alpha_1 - \alpha_2 = 2\beta_2 y_0$$

$$y_0 = \frac{\alpha_1 - \alpha_2}{2\beta_2} = c \tag{4.11}$$

With the maximum point and y-intercept defined, the slope can now be found as,

$$m = \frac{\frac{1-\alpha_2}{2\beta_2} - \frac{\alpha_1-\alpha_2}{2\beta_2}}{\frac{1-\alpha_1}{2\beta_1}}$$

$$m = \frac{(1-\alpha_1)/2\beta_2}{(1-\alpha_1)/2\beta_1}$$

$$\therefore \; m = \frac{\beta_1}{\beta_2} \tag{4.12}$$

This gives the completed Equation 4.13 for the optimum screen feed on to screen B, for a given feed to screen A, shown as the magenta line in Figure 4.16.

$$y = \frac{\beta_1}{\beta_2}x + \frac{\alpha_1 - \alpha_2}{2\beta_2} \tag{4.13}$$

The surface shown in Figure 4.16 can then be projected onto new axes in terms of a ratio, r, of feed to Screen A, as a proportion of total feed M_t.

Letting $x = M_t r$ and $y = M_t(1 - r)$, and substituting into Equation 4.4, yeilds the transformed surface shown in Figure 4.17.

The optimum ratio for any given total feed rate can the be determined by substituting $x = M_t r$ and $y = M_t(1 - r)$, into Equation 4.13, yielding,

$$M_t(1 - r) = \frac{\beta_1}{\beta_2} M_t r + \frac{\alpha_1 - \alpha_2}{2\beta_2} \tag{4.14}$$

After isolating the r term on the LHS, the equation becomes:

$$r = \frac{M_t - \frac{\alpha_1-\alpha_2}{2\beta_2}}{M_t(1 + \beta_1/\beta_2)} \tag{4.15}$$

Giving a formula for the optimum screen feed split at a given total feed rate in terms of the coefficients of fine screen efficiency. The specific optimum path for the example case given, follows the magenta line in Figure 4.17.

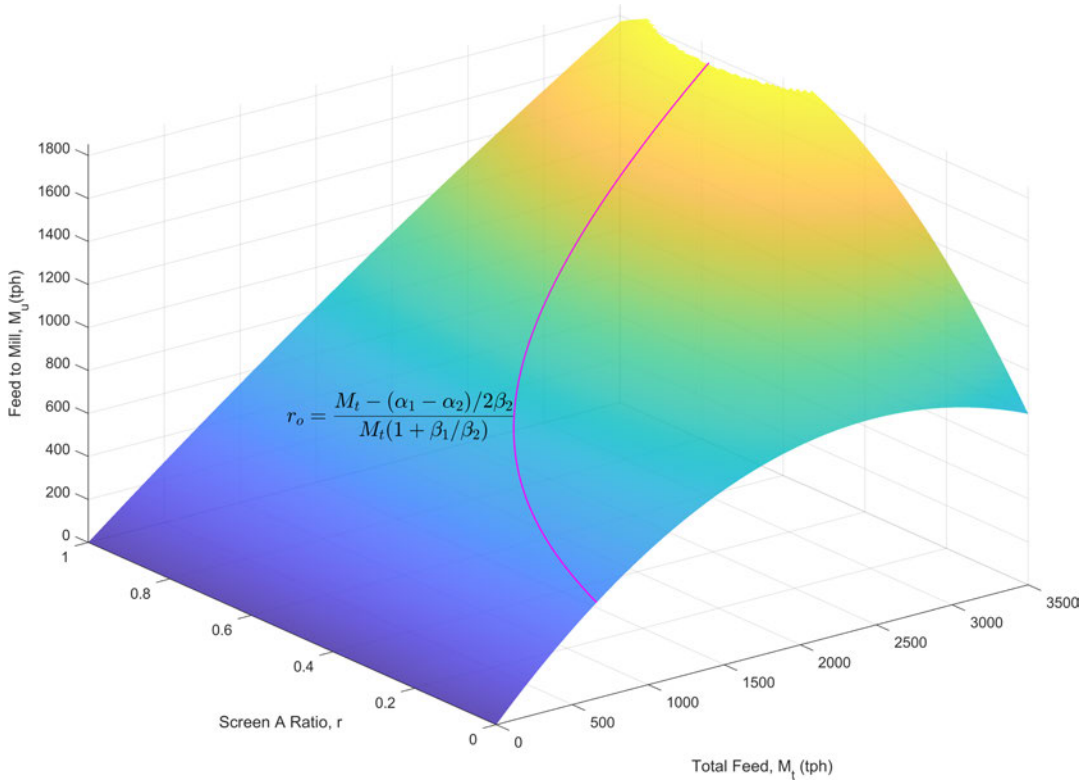

Figure 4.17: Effect of changing the screen feed ratio, r, on mill feed for a given total feed to a pair of screens, M_t. The magenta line plots the optimum ratio, r_o, that maximises M_t at any given feed rate, which is a function of the model coefficicents.

4.4.3 Discussion

Provided that there is some certainty around the accuracy of the regressed coefficients of α_n and β_n, the above analysis highlights a clear relationship between screen feed ratio, efficiency and available mill feed. This ultimately demonstrates a viable method to optimise the screen feed rates based on the model. Importantly, an equation for the optimum ratio (Eq. 4.15) was derived that would be relatively straightforward to implement in the DCS. While not all screen pairs would exhibit such strong curvature, this particular test case indicates that somewhat aggressive control of the ratio may be required, starting from all feed onto screen B initially and then pivoting towards 70% feed onto screen B at higher total feed-rates. Such an implementation would require careful management of the bounds and rate-of-change to safeguard the process from adrupt disturbances.

Also not included in this analysis, is the effect of preferentially feeding one screen over another on screen wear-rates and potential conflicts with existing maintenace strategies. Further investigation into this relationship would need to be carried out to ensure that that

there is a net positive benefit in implementing the optimising control strategy. Regardless, the simulated response provides good justification for continued development.

4.5 Alarm and Monitoring

At the time of preparing this paper, the alarm and monitoring scheme had not been completed. While time was a factor, the stability of the current solution still leaves room for improvement, making it challenging to determine baseline performance. Additionally, field verifications will need to take place so that correct conclusions can be drawn between increased predicted oversize and actual screen performance. With these matters addressed, the implementation of such a scheme would still prove beneficial.

4.6 Chapter Summary

This chapter has provided comprehensive detail on the implementation of an online parameter estimation scheme, and discussed the effective verification of the model through simulation. While the intended field verification activity prescribed in Section 3.4 unfortunately did not take place, the analysis performed still served as a thorough assessment, and highlighted several opportunities for refinement. In addition to this, a detailed analysis of how modifying the feed ratio can be used to optimise total mill feed has been provided, culminating in the derivation of an optimising algorithm that, given a stable model, is ready for DCS implementation.

Chapter 5

Conclusions and Further Work

5.1 Conclusions

This research project set out to address the following major objectives:

1. Review existing theory, models and methods for determining fine screen efficiency.
2. Define the structure of the model that best represents the system.
3. Research and compare algorithms used for parameter estimation, in the context of the defined model structure and fine screen efficiency in general.
4. Using the selected algorithm(s), develop an application to identify the parameters of the system model online, as a measure of fine screen efficiency.
5. Verify the performance of the model against actual plant operation.
6. Use the derived model to assess the impact of adjusting the fine screen feed ratio on mill throughput.

In addition to defining the project objectives, Chapter 1 of this dissertation provided an overview of the plant in which this research took place and the motivations behind the project. Objective 1 was addressed by the literature review in Chapter 2 that discussed, in detail, the theory regarding ore classification alongside a broader overview of linear modeling and techniques for parameter estimation.

Using this information, the methodology in Chapter 3 set out to first define a static fine screen oversize model based on the mass balance of the system. It was shown that this could easily be extended to both a dynamic system and a quadratic model, and how such model fits into the linear regression paradigm, fully addressing Objective 2. Chapter 3 also fully addressed Objective 3, with the comparison and selection of a suitable parameter estimation algorithm, while laying down the groundwork for the development of an online estimating application.

In addressing Objective 4, Chapter 4 thoroughly discussed the complete implementation of the online estimator, with reference to the code listings in Appendix D. In addition to the MATLAB® implementation, the required DCS components were also presented. For both facets, conceptual diagrams and screenshots of the application helped to illustrate the details. Following the main implementation sections, Chapter 4 attempted to verify the performance by observing the response on the DCS with historian trends. This was complemented by an analysis of the models sensitivity to the sliding window size. Although this does not precicely address Objective 5 due to the absence of the stated field verification, the process was still informative and suggested that some work still needs to be done to improve the solution. Despite this, Objective 6 was fully addressed in Section 4.4 and an insightful algorithm to optimise the mill feed based on the model was fully specified.

Other than time constraints, the main challenge to completing this project successfully mostly pertained to an underestimation of the amount of statistical theory that would be required. While it is believed that the model structure is analytically correct, more thought into the input characteristics and techniques to test the suitability of the data would result in better data validation processes. Ultimately, the project would have greatly benefited from a more rigorous statistical analysis overall that was simply beyond the scope of the original objectives.

The ultimate aim of this project, as stated in Section 1.4, was to develop an algorithm to perform the online parameter estimation of a fine screen efficiency model. To this end, and given that in general, the main objectives have been mostly met, this project should be considered a success. While the model itself has presented mixed results, it has clearly been demonstrated that the method to develop an inferential measurement of fine screen efficiency is a feasible one and with some refinements be able to be incorporated in an optimising control strategy.

5.2 Further Work

While most of the major objectives were met, time constraints and lack of knowledge resulted in some omissions and issues that need to be addressed in the future. While the implementation generally does provide good estimates of the fines screen oversize, the stability of the solver still needs to be improved further if the online optimiser described in Section 4.4 were to be implemented.

To improve the model stability and accuracy, the following further work is recommended:

- Investigate the implementation of small excitation of the feed to each screen to generate more variability in input data.
- A more rigorous statistical analysis of the input characteristics and requirements.
- From 2, improve the input validation scheme using more robust statistical techniques.
- Explore other potential estimation algorithms that may be more to immune to high input covariance.
- Explore recursive estimation schemes such as the Kalman Filter.

If the model stability is improved, field verification as per the original Objective 5, can be carried out. Once the model is verified as accurate and predicable, some final works can be completed:

- Implement the online optimisation scheme as per Objective 7.
- Implement an alarm and monitoring scheme, as per Objective 8.
- Refine the application to make it more portable and less reliant on MATLAB®.

References

Atlassian (2021), *What is Agile*, The Agile Coach, Atlassian. viewed 09 Oct 2021, https://www.atlassian.com/agile.

Bergmeir, C., Hyndman, R. J. & Koo, B. (2018), 'A note on the validity of cross-validation for evaluating autoregressive time series prediction', *Computational Statistics & Data Analysis* **120**, 70–83. viewed 23 May 2021, https://www.sciencedirect.com/science/article/pii/S0167947317302384.

Brownlee, J. (2018), 'A gentle introduction to k-fold cross-validation', *Machine Learning Mastery* . viewed 23 Sep 2021, https://machinelearningmastery.com/k-fold-cross-validation/.

Cerqueira, V., Torgo, L. & Mozetič, I. (2020), 'Evaluating time series forecasting models: an empirical study on performance estimation methods', *Machine Learning* **109**(11), 1997–2028. viewed 23 May 2021, https://doi.org/10.1007/s10994-020-05910-7.

Chang, W.-D. (2007), 'Nonlinear system identification and control using a real-coded genetic algorithm', *Applied Mathematical Modelling* **31**(3), 541–550. viewed 01 May 2021, https://www.sciencedirect.com/science/article/pii/S0307904X05002519.

Cliffydcw (2012), *Software Development Life Cycle*, digital image of model, Wikimedia Commons. viewed 09 Oct 2021, https://commons.wikimedia.org/wiki/File:SDLC_-_Software_Development_Life_Cycle.jpg#filelinks.

Coello, C. A. C. & Lechuga, M. S. (2002), 'Mopso: a proposal for multiple objective particle swarm optimization', **2**, 1051–1056 vol.2. viewed 02 May 2021, https://ieeexplore.ieee.org/document/1004388.

Control Station (2018), *What is a Distributed Control System?*, blog post, Control Station. viewed 11 October 2021, https://controlstation.com/blog/what-is-a-distributed-control-system/.

Dangprasert, P. & Avatchanakorn, V. (1996), 'Genetic algorithms based on an intelligent controller', *Expert systems with applications* **10**(3), 465–470. viewed 10 Apr 2021, https://www-sciencedirect-com.ezproxy.usq.edu.au/science/article/pii/0957417496000267.

Darlington, R. B. & Hayes, A. F. (2016), *Regression Analysis and Linear Models: Concepts, Applications, and Implementation*, Guilford Publications, New York, UNITED STATES. viewed 23 May 2021, http://ebookcentral.proquest.com/lib/usq/detail.action?docID=4652287.

De Veaux, R. D., Velleman, P. F. & Bock, D. E. (2016), *Stats: Data and Models, Global Edition*, 4 edn, Pearson.

Dobson, A. J. & Barnett, A. G. (2018), *An Introduction to Generalized Linear Models*, CRC Press LLC, Milton, UNITED KINGDOM. viewed 24 May 2021, http://ebookcentral.proquest.com/lib/usq/detail.action?docID=5488477.

Fister, D., Fister, I., Fister, I. & Šafarič, R. (2016), 'Parameter tuning of pid controller with reactive nature-inspired algorithms', *Robotics and Autonomous Systems* **84**, 64–75. viewed 19 Apr 2021, https://www.sciencedirect.com/science/article/pii/S0921889016301439.

Fleming, P. J. & Purshouse, R. C. (2002), 'Evolutionary algorithms in control systems engineering: a survey', *Control Engineering Practice* **10**(11), 1223–1241. viewed 01 Apr 2021, http://www.sciencedirect.com/science/article/pii/S0967066102000813.

Frost, J. (2017*a*), 'The difference between linear and nonlinear regression models', *Statistics by Jim* . viewed 11 September 2021, https://statisticsbyjim.com/regression/difference-between-linear-nonlinear-regression-models/.

Frost, J. (2017*b*), 'Explained: Neural networks', *MIT News Office* . viewed 11 September 2021, https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

IBM Cloud Education (2020), 'Neural networks', *IBM Cloud Learn Hub* . viewed 11 September 2021, https://www.ibm.com/au-en/cloud/learn/neural-networks.

IEEE (2017), 'ISO/IEC/IEEE international standard - systems and software engineering – software life cycle processes', *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11* pp. 1–157. viewed 09 Oct 2021, https://ieeexplore-ieee-org.ezproxy.usq.edu.au/document/8100771.

Johnson, R. (2012), Matlab and TDD. viewed 23 May 2021, https://www.mathworks.com/matlabcentral/fileexchange/37383-matlab-and-tdd.

Jung, Y. & Hu, J. (2015), 'A k-fold averaging cross-validation procedure', *Journal of Nonparametric Statistics* **27**(2), 167–179. viewed 23 May 2021, https://doi.org/10.1080/10485252.2015.1010532.

King, M. (2016), *Process control : A Practical Approach*, second edition edn, Wiley, Chichester, England.

Kumar, M. & Rashid, E. (2018), 'An efficient software development life cycle model for developing software project', *International Journal of Education and Management Engineering* **8**(6), 59. viewed 09 Oct 2021, https://www.proquest.com/scholarly-journals/efficient-software-development-life-cycle-model/docview/2150541798/se-2?accountid=14647.

Lever, J., Krzywinski, M. & Altman, N. (2016), 'Points of significance: Model selection and overfitting', *Nature Methods* **13**(9), 703–704. viewed 30 May 2021, https://www.proquest.com/scholarly-journals/points-significance-model-selection-overfitting/docview/1817862312/se-2?accountid=14647.

Lewin, D. R. (2005), 'Evolutionary algorithms in control system engineering', *IFAC Proceedings Volumes* **38**(1), 45–50. viewed 30 May 2021, https://www.sciencedirect.com/science/article/pii/S147466701636880X.

Li, X., Sha, J. & Wang, Z.-l. (2016), 'A comparative study of multiple linear regression, artificial neural network and support vector machine for the prediction of dissolved oxygen', *Hydrology Research* **48**(5), 1214–1225. viewed 28 Aug 2021, https://doi.org/10.2166/nh.2016.149.

Mahmoud, M. S., Sabih, M. & Elshafei, M. (2015), 'Using opc technology to support the study of advanced process control', *ISA Trans* **55**, 155–67. viewed 10 Oct 2021, https://www.ncbi.nlm.nih.gov/pubmed/25702044.

Mathworks (2021*a*), 'crossval', *ga* . viewed 11 May 2021, https://au.mathworks.com/help/stats/crossval.html.

Mathworks (2021*b*), 'cvpartition', *ga* . viewed 11 May 2021, https://au.mathworks.com/help/stats/cvpartition.html.

Mathworks (2021*c*), 'fitlm', *Mathworks MATLAB Documentation* . viewed 23 May 2021, https://au.mathworks.com/help/stats/fitlm.html.

Mathworks (2021*d*), 'fitrnet', *ga* . viewed 11 May 2021, https://au.mathworks.com/help/stats/fitrnet.html#.

Mathworks (2021*e*), 'fitrsvm', *ga* . viewed 11 May 2021, https://au.mathworks.com/help/stats/fitrnet.html.

Mathworks (2021*f*), 'Least-squares (model fitting) algorithms', *Mathworks MATLAB Documentation* . viewed 11 May 2021, https://au.mathworks.com/help/optim/ug/least-squares-model-fitting-algorithms.html.

Mathworks (2021*g*), 'lsqlin', *Mathworks MATLAB Documentation* . viewed 11 May 2021, https://au.mathworks.com/help/optim/ug/lsqlin.html.

Mathworks (2021*h*), 'mldivide', *Mathworks MATLAB Documentation* . viewed 11 May 2021, https://au.mathworks.com/help/matlab/ref/mldivide.html.

Mathworks (2021*i*), 'Opc toolbox', *ga* . viewed 11 October 2021, https://au.mathworks.com/help/opc/index.html.

Mathworks (2021*j*), 'particalswarm', *Global Optimization Toolbox* . viewed 11 May 2021, https://au.mathworks.com/help/gads/particleswarm.html.

Mathworks (2021*k*), 'particalswarm', *ga* . viewed 11 May 2021, https://au.mathworks.com/help/gads/ga.html.

Mathworks (2021*l*), 'polyfit', *Mathworks MATLAB Documentation* . viewed 11 May 2021, https://au.mathworks.com/help/matlab/ref/polyfit.html.

Mathworks (2021*m*), 'Ways to write unit tests', *Mathworks MATLAB Documentation* . viewed 23 May 2021, https://au.mathworks.com/help/matlab/matlab_prog/ways-to-write-unit-tests.html.

Mathworks (2021*n*), 'What is MATLAB?', *ga* . viewed 11 October 2021, https://au.mathworks.com/discovery/what-is-matlab.html.

Newmont Corporation (2021), 'Operations & Projects, Boddington'. viewed 20 May 2021, https://www.newmont.com/operations-and-projects/global-presence/australia/boddington-au/default.aspx.

OPC Foundation (n.d.), 'What is OPC?'. viewed 11 October 2021, https://opcfoundation.org/about/what-is-opc/.

Pramoditha, R. (2020), 'k-fold cross-validation explained in plain english', *Towards Data Science* . viewed 23 May 2021, https://towardsdatascience.com/k-fold-cross-validation-explained-in-plain-english-659e33c0bc0.

Rencher, A. C. & Christensen, W. F. (2012), *Methods of Multivariate Analysis*, John Wiley & Sons, Incorporated, Somerset, UNITED STATES. viewed 23 Jun 2021, http://ebookcentral.proquest.com/lib/usq/detail.action?docID=875890.

Renders, J. M., Nordvik, J. P. & Bersini, H. (1992), 'Genetic algorithms for process control: A survey', *IFAC Proceedings Volumes* **25**(10), 323–328. viewed 17 Apr 2021, https://www.sciencedirect.com/science/article/pii/S1474667017508411.

Rotich, N., Tuunila, R., Elkamel, A. & Louhi-Kultanen, M. (2016), 'Dynamic population balance and flow models for granular solids in a linear vibrating screen', *AIChE Journal* **62**(11), 3889–3898. viewed 22 May 2021, https://doi-org.ezproxy.usq.edu.au/10.1002/aic.15318.

Shaffer, B. (2020), 'QR Matrix Factorization', *Towards Data Sscience* . viewed 22 Jun 2021, https://towardsdatascience.com/qr-matrix-factorization-15bae43a6b2.

Shanmugam, B. K., Vardhan, H., Raj, M. G., Kaza, M., Sah, R. & Hanumanthappa, H. (2021), 'Artificial neural network modeling for predicting the screening efficiency of coal with varying moisture content in the vibrating screen', *International Journal of Coal Preparation and Utilization* pp. 1–19. viewed

22 Jun 2021, https://www-tandfonline-com.ezproxy.usq.edu.au/doi/pdf/10.1080/19392699.2021.1871610?needAccess=true.

Sharp, T. (2020), 'An Introduction to Support Vector Regression (SVR)', *Towards Data Science* . viewed 23 May 2021, https://towardsdatascience.com/an-introduction-to-support-vector-regression-svr-a3ebc1672c2.

Standards Australia (2017), 'Management of alarm systems for the process industries', *AS IEC 62682:2017* . viewed 13 Oct 2021, https://au.i2.saiglobal.com/management/display/index/0/1185139.

Sullivan, J. (2012), *Screening Theory and Practice*, Triple/S Dynamics, Inc. viewed 20 May 2021, https://www.sssdynamics.com/wp-content/uploads/2018/01/screeningtheory.pdf.

Vanwinckelen, G. & Blockeel, H. (2012), 'On estimating model accuracy with repeated cross-validation', *BeneLearn 2012: Proceedings of the 21st Belgian-Dutch Conference on Machine Learning* pp. 39–44. viewed 23 May 2021, https://limo.libis.be/primo-explore/fulldisplay?docid=LIRIAS1655861&context=L&vid=Lirias&search_scope=Lirias&tab=default_tab&lang=en_US.

Weisberg, S. (2013), *Applied Linear Regression*, John Wiley & Sons, Incorporated, Somerset, UNITED STATES. viewed 22 Jun 2021, http://ebookcentral.proquest.com/lib/usq/detail.action?docID=1574352.

Wills, B. A. & Finch, J. (2015), *Wills' Mineral Processing Technology : An Introduction to the Practical Aspects of Ore Treatment and Mineral Recovery*, Elsevier Science & Technology, Oxford, United Kingdom. viewed 21 May 2021, http://ebookcentral.proquest.com/lib/usq/detail.action?docID=4003232.

Yokogawa (n.d.*a*), 'CENTUM OPC Server (Exaopc)'. viewed 11 October 2021, https://www.yokogawa.com/au/solutions/products-platforms/solution-based-software/data-connectivity/centum-opc-server-exaopc/#Details__Functions.

Yokogawa (n.d.*b*), 'Distributed Control System (DCS)'. viewed 11 October 2021, https://www.yokogawa.com/au/solutions/products-platforms/control-system/distributed-control-systems-dcs/#Details.

Zelinka, I. & Chen, G. (2018), *Evolutionary Algorithms, Swarm Dynamics and Complex Networks : Methodology, Perspectives and Implementation*, Emergence, Complexity and Computation, 26, 1st ed. 2018. edn, Springer Berlin Heidelberg, Berlin, Heidelberg. viewed 22 Apr 2021, https://doi-org.ezproxy.usq.edu.au/10.1002/aic.15318.

Zhang, B., Gong, J., Yuan, W., Fu, J. & Huang, Y. (2016), 'Intelligent prediction of sieving efficiency in vibrating screens', *Shock and Vibration* **2016**, 9175417. viewed 10 Oct 2021, https://doi.org/10.1155/2016/9175417.

Zhang, Y., Wang, S. & Ji, G. (2015), 'A comprehensive survey on particle swarm optimization algorithm and its applications', *Mathematical Problems in Engineering* **2015**, 931256. viewed 30 Mar 2021, https://doi.org/10.1155/2015/931256.

Appendix A

Project Specification

ENG4111/4112 Research Project
Project Specification

For: Alex Kapor

Title: *Apply Advanced Process Control to Fine Screening Circuit At Large Mineral Processing Plant*

Major: Instrumentation, Control & Automation

Supervisor: Catherine Hills
Charlie Fabianto, Newmont Mining Corporation

Enrolment: ENG4111 – EXT S1, 2021
ENG4112 – EXT S2, 2021

Project Aim: To model the performance of eight parallel fine screens by developing a method to calculate the efficiency of each screen in real-time, and to use this information to optimise the circuit and ultimately improve mill throughput.

Programme: **Version 2, 8/04/2021**

1. Define the model requirements and its structure based on the physical system.
2. Review plant historical data alongside existing control and measurement strategies.
3. Review literature for existing approaches to measurement of fine screen efficiency
4. Define the key performance indicators (KPI) of a successful optimization method.
5. Define data requirements and implement any required changes to plant historian and OPC interfaces.
6. Research and compare techniques for extracting multiple variables from noisy data and select one or more appropriate algorithms.
7. Develop a computational model of the real-time individual fine screen efficiency
8. Verify performance of the model against the actual plant operation.
9. Use model to assess how adjusting the fine screen feed ratio will impact total throughput and develop an algorithm for calculating the optimal adjustment.

If time and resources permit…

10. Provided that the accuracy of the solution is acceptable, implement online optimiser (in DCS)
11. Measure and report plant performance against KPIs.
12. Implement alarming and monitoring scheme for fine screen efficiency to inform maintenance and operations of any deterioration in performance.
13. Deploy program as a portable package, complete with a set of tools modules and guides for future use by onsite engineering staff.

Appendix B

Risk Assessment

University of Southern Queensland Print View

USQ Safety Risk Management System

Version 2.0

Safety Risk Management Plan

Risk Management Plan ID:	Status:	Current User:	Author:	Supervisor:	Approver:
RMP_2021_5635	*Approval Requested*	[redacted]	[redacted]	[redacted]	[redacted]

Assessment Title:	ERP2021 Apply APC to Fine Screening Circuit at Large Mineral Processing Plant	Assessment Date:	3/06/2021
Workplace (Division/Faculty/Section):	204070 - School of Mechanical and Electrical Engineering	Review Date:	3/06/2022 (5 years maximum)

Approver:	*Supervisor: (for notification of Risk Assessment only)*
Catherine Hills	*Catherine Hills*

Context

DESCRIPTION:

What is the task/event/purchase/project/procedure?	Research on the application of parameter estimation techniques to solve a process control problem		
Why is it being conducted?	Undergraduate Research Project		
Where is it being conducted?	Newmont Boddington Gold Mine		
Course code (if applicable)	ENG4111	Chemical Name (if applicable)	

WHAT ARE THE NOMINAL CONDITIONS?

Personnel involved	Alexander Kapor
Equipment	Personal Computer / Vibrating Screens / Conveyors
Environment	Large Mineral Processing Plant, though most work will be office based. Possible some limited field visits.
Other	Project also being completed under company specific risk and change management procedures.
Briefly explain the procedure/process	Data gathering, model development, model verification, implementation/modification to control schemes

Assessment Team - who is conducting the assessment?

Assessor(s):	Alexander Kapor
Others consulted: (eg elected health and safety representative, other personnel exposed to risks)	*Site supervisor*

Risk Matrix

Probability	Consequence				
	Insignificant No Injury 0-$5K	Minor First Aid $5K-$50K	Moderate Med Treatment $50K-$100K	Major Serious Injury $100K-$250K	Catastrophic Death More than $250K
Almost Certain 1 in 2	M	H	E	E	E
Likely 1 in 100	M	H	H	E	E
Possible 1 in 1,000	L	M	H	H	H
Unlikely 1 in 10,000	L	L	M	M	M
Rare 1 in 1,000,000	L	L	L	L	L

Recommended Action Guide	
Extreme	E= Extreme Risk – Task ***MUST NOT*** proceed
High	H = High Risk – Special Procedures Required (Contact USQSafe) Approval by VC only
Medium	M= Medium Risk - A Risk Management Plan/Safe Work Method Statement is required
Low	L= Low Risk - Manage by routine procedures.

Risk Register and Analysis

	Step 1	Step 2	Step 2a	Step 2b	Step 3			Step 4				
	Hazards: From step 1 or more if identified	The Risk: What can happen if exposed to the hazard without existing controls in place?	Consequence: What is the harm that can be caused by the hazard without existing controls in place?	Existing Controls: What are the existing controls that are already in place?	Risk Assessment: Consequence x Probability = Risk Level			Additional Controls: Enter additional controls if required to reduce the risk level	Risk assessment with additional controls: Has the consequence or probability changed?			
					Probability	Risk Level	ALARP		Consequence	Probability	Risk Level	ALARP
	Example											
	Working in temperatures over 35° C	Heat stress/heat stroke/exhaustion leading to serious personal injury/death	catastrophic	Regular breaks chilled water available loose clothing fatigue management policy.	possible	high	No	temporary shade shelters essential tasks only close supervision buddy system	catastrophic	unlikely	mod	Yes
1	Access to scr...	Hearing damage, eye injury, entaglement	Moderate	Site and Area inductions, mandatory PPE, two-way radio communication	Unlikely	Me...	☐	Limit exposure time	Moderate	Rare	Low	☑
2	Data gathering	Incorrect data will lead to incorrect results	Minor	Identify correct data sources and use software correctly	Rare	Low	☑					☐
3	Develop mod...	Poor ergonomics due to extensive time on PC	Minor	Regular breaks	Unlikely	Low	☐	Implement stretching regime and ergonomic work area.	Insignifica	Rare	Low	☑
4	Implement o...	Production losses due to poor implementation	Moderate	Verify simulated results before implementing changes in site DCS	Unlikely	Me...	☐	Use onsite change management procedure	Moderate	Rare	Low	☑

Step 5 - Action Plan (for controls not already in place)

	Additional Controls:	Exclude from Action Plan: (repeated control)	Resources:	Persons Responsible:	Proposed Implementation Date:
1	Limit exposure time	☐	Nil required	Alex Kapor	1/03/2021
3	Implement stretching regime and ergonomic work area.	☐	Nil required	Alex Kapor	1/03/2021
4	Use onsite change management procedure	☐	System available and well utilised onsite	Alex Kapor	1/03/2021

Supporting Attachments

No file attached

Step 6 – Request Approval

Drafters Name: Alex Kapor Draft Date: 3/06/2021

Drafters Comments:

Assessment Approval: All risks are marked as ALARP

Maximum Residual Risk Level Low - Manager/Supervisor Approval Required

Document Status: Approval Requested

Step 6 – Approval

Approvers Name: Catherine Hills Approvers Position Title:

Approvers Comments:

I am satisfied that the risks are as low as reasonably practicable and that the resources required will be provided.

Approval Decision: Approve / Reject Date: Document Status: Approval Requested

Appendix C

Ethical Clearance

There are no Ethical Clearances applicable to this project.

Appendix D

Code Listings

All of the following code listings were written by the researcher in MATLAB®.

Cross-Validation Script

The script file `validateModels.m` executes k-fold cross-validation of the candidate models. This was used during the performance evaluation phased of the research project.

Listing D.1: validateModels.m (Script)

```
% validateModels.m
%
% Cross Validate Models
% Alex Kapor 31/07/2021
% This script uses k-fold cross validation to validate the performance
% of each algorithm in solving the model of fine screen oversize ratio.
%
% It requires an appropriate dataset from the PI historian data archive
% and function handles for each algorithm to be tested.
%
% Required toolboxes:
%   - Statistics and Machine Learning Toolbox
%   - Parallel Computing Toolbox

% Clears workspace and closes all windows.
clear; close all; clc;

% Check output folders exist
projectPath = [userpath,'\Project'];
outdir  = [projectPath, '\output\'];

if ~exist(outdir, 'dir')
   mkdir(outdir)
end

% Use saved pi data
%predictorSet (108088x8), responseSet (108088x1)
load('data\sanitisedCVdata_20210630_200d.mat');

% Models to test
CvMdl = struct('name',    { "Linear Using mldivide", ...
                            "Linear Using lsqlin", ...
                            "Linear Using PSO", ...
```

```
                            "Linear Using GA", ...
                            "Quadratic Using lsqlin", ...
                            "Quadratic Using PSO", ...
                            "Quadratic Using GA"
                          }, ...
                'handle', { @cv_linear_mldivide, ...
                            @cv_linear_lsqlin, ...
                            @cv_linear_pso, ...
                            @cv_linear_ga, ...
                            @cv_quad_lsqlin, ...
                            @cv_quad_pso, ...
                            @cv_quad_ga
                            } ...
                        );
longestName = max(strlength([CvMdl.name]));

% Generate crosss-validation partitions. Use same seed for repeatability.
rng(1986);
NUM_OF_FOLDS = 10;
CvPartition = cvpartition(length(predictorSet), 'Kfold', NUM_OF_FOLDS);

% Set up parallel pool
p = gcp;

% Set options for parallel cross-validation using same
% rand substreams to maintain reproducibility.
s = RandStream('mlfg6331_64', 'seed', 1986);
opts = statset('UseParallel', true, 'Streams', s, 'UseSubstreams', true);
selected = 1:2; %numel(CvMdl)
% Test cross validate all models, and save relevant results.
for i = selected
    fprintf('Cross-Validation in progress (%-*s)...', longestName, ...
                                                      CvMdl(i).name);

    % Actual validation
    results = crossval(CvMdl(i).handle, predictorSet, responseSet, ...
                       'Partition', CvPartition, 'Options', opts);

    % To pass multiple results from models back through crossval function,
    % cell arrays must be used. Made a bit tidier by adding to structure.
    CvMdl(i).mse = [results{:,1}];
    CvMdl(i).duration = [results{:,2}];
    CvMdl(i).coefficients = results{:,3};
```

```
77
78      % Summary Statistics
79      CvMdl(i).mse_ave = mean(CvMdl(i).mse);
80      CvMdl(i).mse_std = std(CvMdl(i).mse);
81      CvMdl(i).duration_ave = mean(CvMdl(i).duration);
82      CvMdl(i).duration_std = std(CvMdl(i).duration);
83
84      fprintf('Done\n');
85  end
86
87  fprintf('\nK-fold Cross-Validation Results (K=%d, n=%d):\n\n', ...
88                      NUM_OF_FOLDS, CvPartition.TrainSize(1));
89
90  % Summarise and export results
91  varNames = {'Name', 'Mean RMSE', 'Std Dev of RMSE', ...
92              'Mean Exec Time (s)', 'Std Dev of Exec Time'};
93
94  summary = table([CvMdl(selected).name]', ...
95                  [CvMdl(selected).mse_ave]', ...
96                  [CvMdl(selected).mse_std]', ...
97                  [CvMdl(selected).duration_ave]', ...
98                  [CvMdl(selected).duration_std]', ...
99                  'VariableNames',  varNames);
100 disp(summary);
101 writetable(summary, [outdir,'CvSummary.xlsm']);
102 save([outdir, 'CvResults.mat'], 'summary', 'CvMdl');
```

Model Comparison Script

The script file `compareModels.m` was used to compare the two best static models derived from cross-validation. This script simply trains two competing models on the whole static data set and generates a set of figures for each in order to visually inspect and compare performance.

Listing D.2: compareModels.m (Script)

```
% compareModels.m
% Trains two competing models and then compares them graphically
%
% Alex Kapor 31/07/2021
% This script trains two competing models on the full data set and
% generates a set of figures in order to compare their performance.
%
% It requires an appropriate dataset from the PI historian data archive.
%
% Required toolboxes:
%   - Statistics and Machine Learning Toolbox
clear; close all; clc;

% Check output folders exist
projectPath = [userpath,'\Project\'];
mdl1dir = [projectPath, 'vectors\static_linear\'];
mdl2dir = [projectPath, 'vectors\static_quad\'];
outdir  = [projectPath, 'output\'];
if ~exist(mdl1dir, 'dir')
   mkdir(mdl1dir)
end

if ~exist(mdl2dir, 'dir')
   mkdir(mdl2dir)
end

if ~exist(outdir, 'dir')
   mkdir(outdir)
end
saveExt = '.pdf';

```

```
% Use saved pi data
load('data\sanitisedCVdata_20210630_200d.mat');
%predictorSet (108088x8), responseSet (108088x1)

% Generate figure to visualise data set
figure('Name', 'Test Data', 'units', 'normalized', ...
                    'position', [0.1 0.1 0.5 0.8]);
% Plot of all predictor variable observations
topleft = 1:8*3;
topleft(3:3:end) = [];
subplot(9,3,topleft);
s = stackedplot(predictorSet);
s.DisplayLabels = {'FS101', 'FS102', 'FS201', 'FS202', ...
                   'FS301', 'FS302', 'FS401', 'FS402'};
s.Title = {'Predictor Variables'};
s.Position = s.Position + [0 -0.015 0 0];

% Plot of response variable observations
p = subplot(9,3,[25 26]);
plot(responseSet);
ylabel(p, 'Total OS');
xlabel(p, 'Observations');
title('Response Variable');
p.Position = p.Position + [0 -0.025 0 0];

% Generate a histogram for each variable
% Predictor variables
hAx = gobjects(9,1);
ymax = 0;
for idx = 1:8
    hAx(idx) = subplot(9,3,idx*3);
    h = histogram(predictorSet(:,idx), 31);
    ymax = max([ymax h.Values]);
    hAx(idx).Position = hAx(idx).Position + [0 -0.015 0 0];
end

% Response variable
hAx(9) = subplot(9,3,9*3);
h = histogram(responseSet, 31);

% Set all predictor histograms to same x range.
% set ALL histograms to have the same y range.
ymax = max([ymax h.Values]);
```

```
xmax = round(max(predictorSet,[], 'all')+50, -2);
ymax = round(ymax+500,-3);
xlim(hAx(1:8), [0 xmax]);
ylim(hAx(1:9), [0 ymax]);
hAx(9).Position = hAx(9).Position + [0 -0.025 0 0];
sgtitle('Overview of Training Data');

% Train models
% Model 1
lb1 = zeros(8,1);
ub1 = ones(8,1);
coeff1 = lsqlin(predictorSet, responseSet, [], [], [], [], lb1, ub1);
model(1) = RecircModel(coeff1, @(X, b) X*b, "Linear", ...
                predictorSet, responseSet);
%Model 2
lb2 = zeros(16,1);
ub2 = ones(16,1);
predictors_quad = [predictorSet predictorSet.^2];
coeff2 = lsqlin(predictors_quad, responseSet, [], [], [], [], lb2, ub2);
model(2) = RecircModel(coeff2, @(X, b) [X X.^2]*b, "Quadratic", ...
                predictorSet, responseSet, @(X, b) b(1:8)' + X.*b(9:16)');

% Generates all figures
f1 = visualiseModel(model(1));
f2 = visualiseModel(model(2));

% Export graphics and results table
graphtitles = {'scatter', 'recirc', 'residuals', 'histogram'};
for idx = 1:4
    fname1 = [mdl1dir, graphtitles{idx}, saveExt];
    fname2 = [mdl2dir, graphtitles{idx}, saveExt];
    set(f1(idx), 'renderer', 'painters');
    set(f2(idx), 'renderer', 'painters');
    exportgraphics(f1(idx), fname1);
    exportgraphics(f2(idx), fname2);
end

summaryTbl = table([model(:).name]', [model(:).rmse]', [model(:).bias]', ...
                    'VariableNames', {'Model', 'RMSE', 'Bias'});

writetable(summaryTbl, [outdir, 'two_models_compared.xlsm']);
save([outdir, 'compareModels.mat'], 'model', 'summaryTbl');
```

Recirc Model Class Definition

The class definition `RecircModel.m` is a simple custom class built to encapsulate the trained models for easier analysis. The class has properties for the model functions, coefficients, observations and predictions, and subsequently calculates several statistics of interest.

Listing D.3: RecircModel.m (Class)

```
classdef RecircModel
    %RECIRCMODEL Class to defiine a recirc model and it's coefficents.
    % Also calculates predictions statistics if provided with predictor
    % response data.
    %   Alex Kapor 31/07/2021

    properties
        name string = "Unnamed Model";
        coefficients double
        recirc_func function_handle = @(X, b) b(1:8)' + X.*0';
        mdl_func function_handle = @(X, b) X*b;
        predictors double
        actual_response double
        predicted_response double
        residuals double
        residuals_std double   % standardized
        rmse double
        bias double
        n double
    end

    methods
        function obj = RecircModel(b, f, varargin)
            %RECIRCMODEL Construct an instance of this class
            %   RECIRC(b, f)
            %   Construct with coefficents (b) and model function (f) only.
            %
            %   RECIRC(b, f, predictors, response)
            %   As above plus predictor and response matrices
            %
            %   RECIRC(b, f, predictors, response, rf)
```

```
32              %   As above, plus function handle to calculate screen recirc.
33              obj.coefficients = b;
34              obj.mdl_func = f;
35              if nargin >= 3
36                  obj.name = varargin{1};
37              end
38              if nargin >= 5
39                  obj.predictors = varargin{2};
40                  obj.actual_response = varargin{3};
41                  obj.predicted_response = predict(obj);
42                  obj.n = length(obj.predicted_response);
43                  obj.residuals = getResiduals(obj);
44                  obj.residuals_std = standardize(obj);
45                  [obj.rmse, obj.bias] = stats(obj);
46              end
47              if nargin == 6
48                  obj.recirc_func = varargin{4};
49              end
50          end
51
52          function prediction = predict(obj, varargin)
53              %PREDICT(X) Predict response for given predictors
54              %   Always uses stored coefficents.
55              %   If no arguments passed, uses stored predictor values.
56              if nargin > 1
57                  prediction = obj.mdl_func(varargin{1}, obj.coefficients);
58              else
59                  prediction = obj.mdl_func(obj.predictors, obj.coefficients);
60              end
61          end
62
63          function res = getResiduals(obj)
64              %GETRESIDUALS Returns residuals for stored
65              % prediction  vs response variables
66              res = obj.actual_response - obj.predicted_response;
67          end
68
69          function recirc = getRecirc(obj, varargin)
70              % Calculated recirc for each screen and observation
71              % (Always same dimensions as 'obj.predictors')
72              if nargin > 1
73                  pred = varargin{1};
74              else
```

```
75                  pred = obj.predictors;
76              end
77
78              recirc = zeros(size(pred)) + ...
79                              obj.recirc_func(pred, obj.coefficients);
80          end
81          % Statistics
82          function [RMSE, bias] = stats(obj)
83              RMSE = sqrt(sum(obj.residuals.^2)./obj.n);
84              bias = mean(obj.residuals);
85          end
86          function res_s = standardize(obj)
87              std_s = std(obj.residuals);
88              res_s = obj.residuals./std_s;
89          end
90      end
91  end
```

Visualise Model Function

The MATLAB® function `visualseModel.m` is a function that accepts objects of the `RecircModel` class and outputs a set of figures for graphical analysis.

Listing D.4: visualiseModel.m (Function)

```
function figure_hdls = visualiseModel(rmodel)
%VISUALISE_MODEL Generates figures to help visualise model fitness.
% Alex Kapor 31/01/2021

arguments
    rmodel RecircModel % Requires object of RecircModel custom class
end

% Plot labels in LaTex format
labels = {'$\beta_{1}$ (FS101)', '$\beta_{2}$ (FS102)', ...
          '$\beta_{3}$ (FS201)', '$\beta_{4}$ (FS202)', ...
          '$\beta_{5}$ (FS301)', '$\beta_{6}$ (FS302)', ...
          '$\beta_{7}$ (FS401)', '$\beta_{8}$ (FS402)'};

% Vector of figure handle objects for external use
figure_hdls = gobjects(5,1);

% Scatterplot of Predicted Resonse vs Actual Response
figure_hdls(1) = figure('Name', rmodel.name, 'units', 'normalized', ...
                    'position', figPos(0.575, 16/9));
scatter(rmodel.actual_response, rmodel.predicted_response);
xlim([0 7000]);
ylim([0 7000]);
hold on;
x = [0 7000];
plot(x,x);
hold off;
title('Predicted Total Oversize');
xlabel('Actual (tph)');
ylabel('Predicted (tph)');
get(gcf, 'Position')

% Plot of modelled oversize ratio, with calculated coefficents
```

```
figure_hdls(2) = figure('Name', rmodel.name, 'units', 'normalized', ...
                    'position', figPos(0.575, 16/9));
set(gcf,'PaperPositionMode','auto')
subplot(1,3,[1 2]);
x = (0:100:1800)'.*ones(1,8);
plot(x, rmodel.getRecirc(x), 'LineWidth', 0.75);
hT = legend(labels, 'Location', 'northwest', 'Interpreter', 'latex');
set(hT, 'FontSize', 12)
xlim([0 1800]);
ylim([0 1]);
title('Estimated Oversize Ratio (\beta_{n})');
xlabel('Feedrate (tph)');
ylabel('\beta_{n}');
get(gcf, 'Position')

% Display calculated coefficents ($\beta_{n}$)
subplot(1,3,3);
strc = cell(8,1);
lfmt = '$\\beta_{%1d} = %1.3f$';
qfmt = '$\\beta_{%1d} = %1.3f + %1.3f\\sc{E}^{%03d} x_{%1d}$';
for idx = 1:8
    % Pure linear model
    if length(rmodel.coefficients) == 8
        strc{idx} = sprintf(lfmt, idx, rmodel.coefficients(idx));
    % Quadratic terms
    elseif length(rmodel.coefficients) == 16
        exponent = floor(log10(rmodel.coefficients(idx+8)));
        base = rmodel.coefficients(idx+8)/(10^exponent);
        strc{idx} = sprintf(qfmt, idx, rmodel.coefficients(idx), ...
                                                base, exponent, idx);
    else
    strc{1} = 'Unexpected num. of coeff.';
    end
end

h = text(-0.2, 0.50, strc, 'FontName', 'FixedWidth', ...
                    'FontSize', 12, ...
                    'HorizontalAlignment', 'left', ...
                    'Interpreter', 'latex');
h.FontSize = 12;
axis off;
get(gcf, 'Position')

```

```
77 % Scatter plot of residuals
78 figure_hdls(3) = figure('Name', rmodel.name, 'units', 'normalized', ...
79                     'position', figPos(0.3, 4/3));
80 scatter(rmodel.actual_response, rmodel.residuals);
81 % Any gradient between residuals and response may indicate non-linearities.
82 lin = fitlm(rmodel.actual_response, rmodel.residuals);
83 x = [0 7000]';
84 y = predict(lin, x);
85 hold on;
86 plot(x,y);
87 hold off;
88 ylim([-1000 1000]);
89 title('Residuals Plot');
90 xlabel('Actual Total Oversize (tph)');
91 ylabel('Residual')
92 get(gcf, 'Position')
93
94 % Histogram of Residuals
95 figure_hdls(4) = figure('Name', rmodel.name, 'units', 'normalized', ...
96                     'position', figPos(0.3, 4/3));
97 histogram(rmodel.residuals, linspace(-1000,1000, 22));
98 title('Residual Distribution');
99 xlabel('Residual');
100 ylabel('Frequency');
101 get(gcf, 'Position')
102 end
```

Create OPC Read Group

The MATLAB® function `createReadGroup.m` is a method belonging to the main GUI app class, and creates an OPC group for the reading of OPC data. The list of OPC tags is taken from the application workspace and added to the group.

Listing D.5: createReadGroup.m (Class Method)

```
function createReadGroup(app)
    % CREATEREADGROUP Class method to create an OPC group for reading from
    % the DCS and adding the tags to the group. Group starts as 'inactive',
    % and must be enabled seperately by calling app after OPC connection
    % is established.
    %
    % Once enabled, read operations are asynchronous at 'OpcInputRate'
    % intervals.
    %
    % Function can only be called from the main FSOS GUI app.
    %
    % Alex Kapor 2021
    %

    % Remove any group alread initialised
    app.deleteGroup(app.OpcReadGroup)

    % Create new group
    label = "OpcReadGrp";
    app.OpcReadGroup = addgroup(app.OpcDaObj, label);
    app.OpcReadGroup.UpdateRate = app.Cfg.OpcInputRate;

    % Start with group set to inactive
    set(app.OpcReadGroup, 'Active', 'off');

    % Now add items to group
    n = app.UITableInputGroup.Data.Tag ~= "";
    try
        fsos.opc.addItems(app.OpcReadGroup, ...
                          app.UITableInputGroup.Data.Tag(n));
        app.MsgTextArea.Value = sprintf("Items added to '%s'",...
                                                    label);
```

```
33          pause(1);
34          app.InputGroupTab.ForegroundColor = [0,0,0];
35      catch ME
36          msg = 'Unable to add tags to group "%s":\n%s';
37          app.MsgTextArea.Value = sprintf(msg, label, ME.message);
38          app.InputGroupTab.ForegroundColor = [1,0,0];
39          return
40      end
41
42      % Initialise the input buffer object
43      app.InputFilter = fsos.cls.Fopdt(...
44                            'Channels', 9,...
45                            'TimeConstant', 2, ...
46                            'DelaySeconds', 0, ...
47                            'SamplePeriod', app.Cfg.OpcInputRate);
48
49      % Configure update timer
50      set(app.InputTimer, 'Period', app.Cfg.OpcInputRate);
51  end
```

Create OPC Write Group

The MATLAB® function `createWriteGroup.m` is a method belonging to the main GUI app class, and creates an OPC group for the writing of OPC data. The list of OPC tags is taken from the application workspace and added to the group.

Listing D.6: createWriteGroup.m (Class Method)

```
function createWriteGroup(app)
    % CREATEWRITEGROUP Class method to creates an OPC group for writing
    % from the DCS and adding the tags to the group. Group starts as
    % 'inactive', and must be enabled seperately by the calling app after
    % OPC connection is established.
    %
    % Subscription is also turned off as the write operations will be
    % performed ad-hoc and asynchronously.
    %
    % Function can only be called from the main FSOS GUI app.
    %
    % Alex Kapor 2021
    %

    % Remove any group alread initialised
    app.deleteGroup(app.OpcWriteGroup)

    % Create new group
    label = "OpcWriteGrp";
    app.OpcWriteGroup = addgroup(app.OpcDaObj, label);

    % Start with group set to inactive
    set(app.OpcWriteGroup, 'Active', 'off');
    set(app.OpcWriteGroup, 'Subscription', 'off');

    % Now add any non-blank tags to group
    n = app.UITableOutputGroup.Data.Tag ~= "";
    try
        fsos.opc.addItems(app.OpcWriteGroup, ...
                        app.UITableOutputGroup.Data.Tag(n));
        app.MsgTextArea.Value = sprintf("Items added to '%s'",...
                                                        label);
```

```
33          pause(1);
34          app.OutputGroupTab.ForegroundColor = [0,0,0];
35      catch ME
36          msg = 'Unable to add tags to group "%s":\n%s';
37          app.MsgTextArea.Value = sprintf(msg, label, ME.message);
38          app.OutputGroupTab.ForegroundColor = [1,0,0];
39          return
40      end
41  end
```

ValidateInput

The MATLAB® class definition `ValidateInput.m` is a class that inherits the matlab.System class, making it a 'System Object'. The class allows the creation of an input validation object that checks the validity/usability of input data. This object acts as a connector between the raw OPC data and the aggregated solver class.

Listing D.7: ValidateInput.m (Class)

```
classdef ValidateInput < matlab.System
    % VALIDATEINPUT This MATLAB System Object determines whether an
    % observation should be used in the model training set.
    %
    % In addition to eliminating outliers, it ensures that the standard
    % deviation across observation predictor variables is high enough
    % as well as whether the response variable is non-zero.
    %
    % Alex Kapor 14-10-2021
    %

    % Private constant properies
    properties(Access = private, Constant)
        pNumOfPredictors = 8
        pNumOfResponse = 1
        pNumChannels = 9;
        pPredictorRange = 1:8;
        pResponseRange = 9;
    end

    % Public, tunable properties
    properties
        SdCutoff (1, 1) {mustBePositive} = 100
        TpCutoff (1, 1) {mustBePositive} = 500
        OutlierCutoff (1, 1) {mustBePositive} = 3
        MaxLowSdCount (1, 1) {mustBePositive, mustBeInteger} = 10
        MaxLowTpCount (1, 1) {mustBePositive, mustBeInteger} = 2
    end

    properties (Nontunable)
        WindowLength (1, 1) {mustBePositive, mustBeInteger} = 10
```

```
32      end
33
34      properties (SetAccess = private)
35          SdScrFd = 0                 % Standardised by the mean
36          MedianScrFd = 0
37          Mean = 0
38          Sd = 0
39          State = '-'
40      end
41
42      properties(DiscreteState)
43          LowTpCount
44          LowSdCount
45          Buffer
46      end
47
48      methods
49          function obj = ValidateInput(varargin)
50              % Support name-value pair arguments when constructing object
51              setProperties(obj,nargin,varargin{:})
52          end
53      end
54
55      methods(Access = protected)
56          function setupImpl(obj)
57              % Set counters
58              obj.LowTpCount = 0;
59              obj.LowSdCount = 0;
60
61              % Initialise Buffer
62              obj.Buffer = NaN(obj.WindowLength, obj.pNumChannels);
63
64          end
65
66          function y = stepImpl(obj, u)
67          % Implement algorithm. Calculate y as a function of input u and
68              % Early return for bad/missing data
69              if ( any(isnan(u)) || ~isequal(size(u), [1 obj.pNumChannels]))
70                  y = [];
71                  obj.State = 'BAD';
72                  return
73              end
74
```

```
75              % Apply moving average filter
76              obj.Buffer = [obj.Buffer(2:end, :); u];
77              obj.Mean = mean(obj.Buffer, 1, 'omitnan');
78              obj.Sd = std(obj.Buffer, [], 1, 'omitnan');
79
80              % Check observation qualities
81              checkForLowSd(obj, obj.Mean(obj.pPredictorRange));
82              checkForLowTp(obj, obj.Mean(obj.pResponseRange));
83              obj.MedianScrFd = mean(u(1:8), 'omitnan');
84
85              % Consider any measurement more than 'OutlierCutoff' standard
86              % deviations away from current mean for that stream to be an outlier.
87              outlier = any(abs(obj.Mean - u) > obj.OutlierCutoff*obj.Sd);
88
89              % Determine whether to pass current input through, or not
90              if (obj.LowSdCount >= obj.MaxLowSdCount)
91                  obj.State = 'LOW_VAR';
92              elseif (obj.LowTpCount >= obj.MaxLowTpCount)
93                  obj.State = 'LOW_FEED';
94              elseif outlier
95                  obj.State = 'OUTLIER';
96              else
97                  obj.State = 'GOOD';
98              end
99
100             % Data is good to go
101             if (isequal(obj.State,'GOOD'))
102                 y = u;
103             else
104                 y = [];
105             end
106         end
107
108         function resetImpl(obj)
109         % Initialize / reset discrete-state properties
110             obj.LowTpCount = 0;
111             obj.LowSdCount = 0;
112             obj.Buffer(:) = NaN;
113         end
114
115         function checkForLowSd(obj, u)
116         % Count low spread events across predictor variables
117         % Spread defined as standardised std. omitting any near zero values.
```

```
118             result = 100*std(u(u>10))/mean(u(u>10));
119             if result < obj.SdCutoff
120                 obj.LowSdCount = min(obj.MaxLowSdCount*2, obj.LowSdCount+1);
121             elseif result > obj.SdCutoff
122                 obj.LowSdCount = max(0, obj.LowSdCount-1);
123             end
124             if ~isnan(result)
125                 obj.SdScrFd = result;
126             else
127                 obj.SdScrFd = -1;
128             end
129         end
130 
131         function checkForLowTp(obj, u)
132         % Count low throughput events on response variable
133             if (u < obj.TpCutoff )
134                 obj.LowTpCount = min(obj.MaxLowTpCount*2, obj.LowTpCount+1);
135             else
136                 obj.LowTpCount = max(0, obj.LowTpCount-1);
137             end
138         end
139     end
140 end
```

Aggregator

The `Aggregator.m` MATLAB® class definition, is another system object that defines an 'Aggregator' whose purpose is to encapsulate the solver objects alongside the model's data buffer. Additionally, the Aggregator allows the mixing of two models, a long-term model and short-term one, through the use of online tunable forgetting factor and model cut parameters.

Listing D.8: Aggregator.m (Class)

```
classdef Aggregator < matlab.System
    % AGGREGATOR Oversees data-fitting process to estimate the
    % coefficients of the plant model. This combines a long term model
    % with a shorter one, using a forgetting factor to perform weighting
    % of the final output.
    %
    % Alex Kapor 14-10-2021

    % Public, tunable properties
    properties
        UpperBounds double {mustBeVector} = 1.0
        LowerBounds double {mustBeVector} = 0.0
        % Weighting of observations in buffer.
        ForgettingFactor double ...
            {mustBeInRange(ForgettingFactor,0,1)} = 0.2
        ModelCut double {mustBeInRange(ModelCut, 0, 1)} = 0.33
    end

    % Public, non-tunable properties
    properties(Nontunable)
        WindowSize (1,1) {mustBePositive,mustBeInteger} = 100000      %(m)
        NumOfPredictors (1, 1) {mustBePositive,mustBeInteger} = 8     %(p)
        NumOfCoefficients (1,1) {mustBePositive,mustBeInteger}  = 16   %(n)
        InitialCoefficients double {mustBeVector} = 0
        InitialData {mustBeNumeric} = []
        NumOfSolvers double {mustBeNumeric} = 2
        PredictFcn = @fsos.mdl.quadPredict
    end

    properties(DiscreteState)
```

```
31          Buffer
32      end
33
34      properties(Access = private)
35          BufferWidth double
36          SolverLong
37          SolverShort
38      end
39
40      %Read only properties
41      properties (SetAccess = private)
42          Coefficients
43          Response
44          Residuals
45          Rmse
46          Bias
47          UpdateTime
48      end
49
50  %% Public Methods
51      methods
52
53          % Constructor
54          function obj = Aggregator(varargin)
55              % Support name-value pair arguments when constructing object
56              setProperties(obj,nargin,varargin{:})
57          end
58
59          %Public user functions
60          function update(obj)
61              thistic = tic();
62              buf = obj.Buffer.all();
63
64              % Only split the buffer if we have more than 10000 samples.
65              if length(buf)>10000
66                  cut = floor(length(buf)*(1 - obj.ModelCut));
67              else
68                  cut = 1;
69              end
70
71              [b1, ~] = obj.SolverLong.fit(buf(:,1:9));
72              [b2, ~] = obj.SolverShort.fit(buf(cut:end,1:9));
73
```

```
74              obj.Coefficients = (1-obj.ForgettingFactor)*b1'+...
75                                          obj.ForgettingFactor*b2';
76
77              % Calculate statistics
78              totals = obj.predictTotalOs(buf(cut:end, 1:8));
79              obj.Response = [buf(cut:end, 9), totals];
80              obj.Residuals =  obj.Response(:,2) - obj.Response(:,1);
81              obj.Rmse = sqrt(mean(obj.Residuals.^2));
82              obj.Bias = mean(obj.Residuals);
83              obj.UpdateTime = toc(thistic);
84          end
85
86          function out = predictTotalOs(obj, u)
87              if isequal(obj.NumOfPredictors, size(u,2))
88                  tmp = obj.predictScreenOs(u);
89                  out = sum(tmp,2);
90              else
91                  error('Incorrect input size passed to predictTotalOs.');
92              end
93          end
94
95          function out = predictScreenOs(obj, u)
96              if isequal(obj.NumOfPredictors, size(u,2))
97                  split = obj.predictSplit(u);
98                  out = u.*split;
99              else
100                 error('Incorrect input size passed to predictScreenOs.');
101             end
102         end
103
104         function out = predictSplit(obj, u)
105             if isequal(obj.NumOfPredictors, size(u,2))
106                 out = obj.PredictFcn(obj.Coefficients, u);
107             else
108                 error('Incorrect input size passed to predictSplit.');
109             end
110         end
111
112         function out = rmse(obj)
113             out = obj.SolverLong.rmse();
114         end
115
116         function out = oldest(obj)
```

```
117             firstrow = obj.Buffer.first();
118             out = now - firstrow(10);
119         end
120 
121         function out = samples(obj)
122             out = obj.Buffer.stored;
123         end
124 
125     end
126 
127     methods(Access = protected)
128         %% Common functions
129         function setupImpl(obj)
130             % Buffer width required is  number of predictors + response
131             % variables plus one to hold a timestamp
132             obj.BufferWidth = obj.NumOfPredictors + 2;
133             if isscalar(obj.UpperBounds)
134                 obj.UpperBounds = repelem(obj.UpperBounds, ...
135                                           obj.NumOfCoefficients);
136             end
137 
138             if isscalar(obj.LowerBounds)
139                 obj.LowerBounds = repelem(obj.LowerBounds, ...
140                                           obj.NumOfCoefficients);
141             end
142 
143             if isscalar(obj.InitialCoefficients)
144                 obj.InitialCoefficients = repelem(obj.InitialCoefficients,...
145                                           obj.NumOfCoefficients);
146             end
147         end
148 
149         function y = stepImpl(obj,u)
150             % Step's main function is just to add new data to the buffer.
151             % Always returns the latest coefficient estimate,
152             % but this is updated through update method.
153 
154             % Append a timestamp to data and push to buffer
155             if ~isempty(u)
156                 u = [u now];
157                 obj.Buffer.push(u);
158             end
159             y = obj.Coefficients;
```

```
160         end
161
162         function resetImpl(obj)
163             % Initialize / reset discrete-state properties
164             obj.Buffer = fsos.cls.CircularBuffer(obj.WindowSize, ...
165                                                  obj.BufferWidth, ...
166                                                  obj.InitialData);
167
168             obj.Coefficients = obj.InitialCoefficients;
169
170             obj.SolverLong = fsos.mdl.Solver(16, @fsos.mdl.quadSolver, ...
171                                              @fsos.mdl.quadPredict, ...
172                                              'LowerBounds', obj.LowerBounds, ...
173                                              'UpperBounds', obj.UpperBounds);
174
175             obj.SolverShort = fsos.mdl.Solver(16, @fsos.mdl.quadSolver, ...
176                                              @fsos.mdl.quadPredict, ...
177                                              'LowerBounds', obj.LowerBounds, ...
178                                              'UpperBounds', obj.UpperBounds);
179             obj.UpdateTime = -1;
180         end
181
182         %% Backup/restore functions
183         function s = saveObjectImpl(obj)
184             % Set properties in structure s to values in object obj
185
186             % Set public properties and states
187             s = saveObjectImpl@matlab.System(obj);
188
189             % Set private and protected properties
190             s.BufferWidth = obj.BufferWidth;
191         end
192
193         function loadObjectImpl(obj,s,wasLocked)
194             % Set properties in object obj to values in structure s
195
196             % Set private and protected properties
197             obj.BufferWidth = s.BufferWidth;
198
199             % Set public properties and states
200             loadObjectImpl@matlab.System(obj,s,wasLocked);
201         end
202
```

```
203         %% Advanced functions
204         function validatePropertiesImpl(obj)
205             % Validate related or interdependent property values
206 
207             % Check initial coefficients vector
208             if ~(isscalar(obj.LowerBounds) || ...
209                     (isvector(obj.LowerBounds) && ...
210                     (obj.NumOfCoefficients == length(obj.LowerBounds))))
211                 error(['Lower bounds must be scalar or vector ',...
212                        'of length equal to number of coefficients']);
213             end
214 
215             if ~(isscalar(obj.UpperBounds) || ...
216                     (isvector(obj.UpperBounds) && ...
217                     (obj.NumOfCoefficients == length(obj.UpperBounds))))
218                 error(['Upper bounds must be scalar or vector ',...
219                        'of length equal to number of coefficients']);
220             end
221 
222             if any(obj.LowerBounds > obj.UpperBounds)
223                 error('Lower bounds must be less than upper bounds');
224             end
225 
226             if ~(isscalar(obj.InitialCoefficients) || ...
227                     (isvector(obj.InitialCoefficients) && ...
228                     (obj.NumOfCoefficients == length(obj.InitialCoefficients))))
229                 error(['Initial coefficient must be scalar or vector ',...
230                        'of length equal to number of coefficients']);
231             end
232 
233             if ~isempty(obj.InitialData) && ...
234                     (size(obj.InitialData, 2) ~= (obj.NumOfPredictors + 2))
235                 error("Initial data incorrect size for buffer")
236             end
237         end
238 
239         function ds = getDiscreteStateImpl(obj)
240             % Return structure of properties with DiscreteState attribute
241             if ~isempty(obj.Buffer)
242                 ds.Buffer = obj.Buffer.all();
243             end
244 
245             ds.Coefficients = obj.Coefficients;
```

```
246             ds.Residuals = {obj.SolverLong.Residuals, ...
247                             obj.SolverShort.Residuals};
248             ds.UpdateTime = obj.UpdateTime;
249 
250         end
251 
252         function processTunedPropertiesImpl(obj)
253             % Perform actions when tunable properties change
254             % between calls to the System object
255             changed = isChangedProperty(obj, 'LowerBounds') || ...
256                         isChangedProperty(obj, 'UpperBounds');
257             if changed
258                 obj.LowerBounds = min(obj.LowerBounds, obj.UpperBounds);
259                 obj.UpperBounds = max(obj.LowerBounds, obj.UpperBounds);
260             end
261         end
262     end
263 end
```

Solver

The `Solver.m` MATLAB® class definition, defines a 'Solver' object whose purpose is to encapsulate a solver instance, its upper and lower bounds, and its most recent coefficients. The actual solver and predictor functions are passed to the Solver class constructor on object instantiation.

Listing D.9: Solver.m (Class)

```
classdef Solver < handle
    %SOLVER Estimates the model parameters using the specified algorithm.
    %   Requires handles to solver and predictor functions,
    %   lower and upper bounds.
    %
    % Alex Kapor 14-10-2021
    %

    % Public properties
    properties
        LowerBounds double {mustBeVector} = -1.0
        UpperBounds double {mustBeVector} = 1.0
        NumOfPredictors (1, 1) {mustBePositive,mustBeInteger} = 8     %(p)
        NumOfCoefficients (1,1) {mustBePositive,mustBeInteger}  = 8   %(n)
        Coefficients
        Residuals
        Rmse
        SolverFcn
        PredictFcn
    end

    % Public methods
    methods
        function obj = Solver(numOfCoeff, solverFcn, predictFcn, varargin)
            % Main constructor

            % Parse name value pairs
            p = inputParser;
            isaFcn = @(x) isa(x,'function_handle');
            checkLength = @(x) isscalar(x) || ...
                                isequal(size(x),[1 numOfCoeff]);
```

```
32              addRequired(p, 'NumOfCoeff', @isscalar)
33              addRequired(p, 'SolverFcn', isaFcn);
34              addRequired(p, 'PredictFcn', isaFcn);
35              addParameter(p, 'LowerBounds', obj.LowerBounds, checkLength);
36              addParameter(p, 'UpperBounds', obj.UpperBounds, checkLength);
37              parse(p, numOfCoeff, solverFcn, predictFcn, varargin{:});
38
39              obj.NumOfCoefficients = p.Results.NumOfCoeff;
40              obj.SolverFcn = p.Results.SolverFcn;
41              obj.PredictFcn = p.Results.PredictFcn;
42
43              % If lower bound argument is scalar, create a vector.
44              obj.LowerBounds = p.Results.LowerBounds;
45              if isscalar(obj.LowerBounds)
46                  obj.LowerBounds = repelem(obj.LowerBounds, ...
47                                            obj.NumOfCoefficients);
48              end
49
50              % If upper bound argument is scalar, create a vector.
51              obj.UpperBounds = p.Results.UpperBounds;
52              if isscalar(obj.UpperBounds)
53                  obj.UpperBounds = repelem(obj.UpperBounds, ...
54                                            obj.NumOfCoefficients);
55              end
56
57          end
58
59          function [b, r] = fit(obj, u)
60              % REGRESS Summary of this method goes here
61              %   Detailed explanation goes here
62
63              % Estimate the model coefficients
64              [obj.Coefficients, obj.Residuals] = obj.SolverFcn(u(:,1:8), u(:,9), ...
65                                                obj.LowerBounds, ...
66                                                obj.UpperBounds);
67
68              % Store coefficients and calculate stats
69              b = obj.Coefficients;
70              r = obj.rmse();
71          end
72
73          function out = predict(obj, u)
74              % PREDICT the oversize ratio for given feed
```

```
75              out = obj.PredictFcn(obj.Coefficients, u);
76          end
77
78          function out = rmse(obj)
79              % RMSE calculate the root mean square error
80              out = sqrt(mean(obj.Residuals.^2));
81          end
82      end
83  end
```

Quadratic Model Solver Function

The MATLAB® function `quadSolver.m` is a function that accepts a set predictor and response observations, and optional upper and lower bounds, and returns the coefficients for the solved quadratic oversize model.

Listing D.10: quadSolver.m (Function)

```
function [beta, residual] = quadSolver(predictors, response, lb, ub)
    % QUADSOLVER Solver quadratic oversize model using contrained LS.
    %   predictors = fine screen feed observations
    %                (time-aligned to total oversize)
    %   response = total oversize observations
    %   lb, ub = Upper and lower coefficient bounds
    %
    % Alex Kapor 01-08-2021
    %

    arguments
        predictors double {mustHaveNColumns(predictors, 8)}
        response double {mustHaveNColumns(response, 1)}
        lb (1, 16) double = -ones(1,16)
        ub (1, 16) double = +ones(1,16)
    end

    assert(size(response, 1) == size(predictors, 1), ...
            "Requires predictor and response to be equal length");
    assert(isequal(size(lb),[1 16]) && isequal(size(ub),[1 16]), ...
            "Lower or upper bound vector dimensions incorrect");

    options = optimoptions('lsqlin', 'Display', 'none');

    % Constrained Least squares with upper and lower bounds
    C = [predictors, predictors.^2];
    d = response;
    [beta,~,residual,~,~,~]  = lsqlin(C, d, [], [], [], [], ...
                                            lb, ub, [], options);
end
```

Quadratic Model Prediction Function

The MATLAB® function `quadPrediction.m` is a function that accepts a vector of eight predictor values (fine screen feed) and a set of 16 coefficients, and returns the predicted oversize ratio based on the quadratic oversize model.

Listing D.11: quadPredict.m (Function)

```
function out = quadPredict(beta, predictors)
    %QUADPREDICT Predict oversize ratio from quadratic model coefficients
    %   beta = Wuadratic oversize model coefficients
    %   predictors = Screen feed rates
    %
    % Alex Kapor 14-10-2021
    %
    arguments
        beta (1,16) double {mustBeNumeric}
        predictors double {mustHaveNColumns(predictors, 8)}
    end

    % Predicted oversize ratio for the current input
    out = beta(1:8) + beta(9:16).*predictors;
end
```

Circular Buffer

The MATLAB® class `CircularBuffer.m` defines a first-in, first-out buffer that minimises copy and move operations when new data is added. This is achieved by holding an index to the oldest row in the matrix, and simply overwriting it when the buffer is full.

Listing D.12: CircularBuffer.m (Class)

```
classdef CircularBuffer < handle
    %CIRCULARBUFFER A fairly basic circular buffer of any size, suitable
    % for buffering 1D vectors of any length or numerical type.
    % Reduces copy operations by overwriting oldest data first.
    %
    % Alex Kapor
    % 04/08/2021
    %
    %  obj = circularBuffer(bufferSize, blockWidth, intitialiser)
    %        bufferSize = Length of buffer. Number of blocks to buffer.
    %        blockWidth = Length of vector to be buffered.
    %        initialiser (optional) = Defaults to NaN(double)
    %
    % Methods:
    % push(data)- Adds a block to the buffer.
    % pop()     - Remove the oldest block from the buffer and return it.
    % first()   - Return data from the oldest block without removing it.
    % last()    - Return data from the newest block without removing it.
    % all()     - Returns the entire buffer in chronological order.
    % at(n)     - Return data from nth oldest block(s)
    % replace(n, data) - Replace nth block with given data
    % isFull()
    % isEmpty()
    %

    properties
        buffer
        len
        blockWidth
        stored
        index
        bufferType = 'double';
```

```
33      end
34
35      methods
36          function obj = CircularBuffer(bufferSize, blockWidth, initialData)
37              %CIRCULARBUFFER Construct an instance of this class
38              %   Detailed explanation goes here
39              arguments
40                  bufferSize
41                  blockWidth
42                  initialData {mustBeNumeric} = []
43              end
44              obj.len = bufferSize;
45              obj.blockWidth = blockWidth;
46              obj = initialise(obj, initialData);
47          end
48
49          function obj = initialise(obj, initialData)
50              %INITIALISE Initialises the buffer with the data
51              obj.bufferType = class(initialData);
52              obj.buffer = nan(obj.len, obj.blockWidth, ...
53                                          obj.bufferType);
54              if ~isempty(initialData)
55                  % Initialise all elements with value and type provided
56                  assert(size(initialData, 2) == obj.blockWidth,...
57                  'Incorrect number of columns on initialisation data.');
58
59                  if size(initialData, 1) > obj.len
60                      warning(['Initialisation data too long for buffer. ',...
61                                  'Some values may have been dropped']);
62                      obj.buffer = initialData((end-obj.len+1):end, :);
63                      obj.stored = obj.len;
64                  else
65                      obj.buffer = initialData;
66                      obj.stored = size(initialData,1);
67                  end
68                  obj.index = obj.stored+1;
69              else
70                  % No initialiser
71                  obj.stored = 0;
72                  obj.index = 1;
73              end
74
75          end
```

```
76
77          function push(obj, block)
78              %PUSH new element onto bottom of queue
79              obj.buffer(obj.index, :) = block;
80              increment(obj);
81          end
82
83          function block = pop(obj)
84              %POP Remove oldest element from the queue and return its value
85              % Sets vacated block to NaN
86              idx = obj.top();
87              block = obj.buffer(idx, :);
88              obj.buffer(idx, :) = nan(1, obj.blockWidth, obj.bufferType);
89              obj.stored = obj.stored - 1;
90          end
91
92          function block = first(obj)
93              %FIRST Get data from oldest element in the queue
94              block = obj.buffer(obj.top(), :);
95          end
96
97          function block = last(obj)
98              %LAST Get data from youngest element in the queue
99              block = obj.buffer(obj.bottom(), :);
100         end
101
102         function blocks = all(obj, varargin)
103             %ALL Return entire buffer contents in correct order
104             % Return all columns or optionally a range of columns.
105             if nargin == 2
106                 columns = varargin{1};
107             else
108                 columns = 1:obj.blockWidth;
109             end
110             stt = obj.top();
111             stp = obj.bottom();
112             if (stt <= stp)
113                 blocks =  obj.buffer(stt:stp, columns);
114             else
115                 blocks = [obj.buffer(stt:end, columns);
116                           obj.buffer(1:stp,   columns)];
117             end
118         end
```

```
119
120         function block = at(obj, n, varargin)
121             %AT Return data at the nth row.
122             % Return all columns or optionally a range of columns.
123             if nargin == 3
124                 columns = varargin{1}; % As range of columns
125             else
126                 columns = 1:obj.blockWidth; % All columns
127             end
128             idx = obj.position(n);
129             block = obj.buffer(idx, columns);
130         end
131
132         function replace(obj, block, n, varargin)
133             %REPLACE data at nth row(s) (and optionally column(s)) with
134             % data given in 'block'.
135             if (nargin == 4)
136                 columns = varargin{1}; % As range of columns
137             else
138                 columns = 1:obj.blockWidth; % All columns
139             end
140             assert(numel(columns)==size(block,2), ...
141                'Column mismatch! Required = %d, Supplied = %d\n', ...
142                 numel(columns), size(block,2));
143
144             idx = obj.position(n);
145             obj.buffer(idx, columns) = block;
146         end
147
148         function bool = isFull(obj)
149             %ISFULL Return boolean true if buffer is full
150             bool = obj.stored==obj.len;
151         end
152
153         function bool = isEmpty(obj)
154             %ISEMPTY Return boolean true if buffer is empty
155             bool = obj.stored==0;
156         end
157     end
158
159     methods (Access = private)
160         function obj = increment(obj)
161             %INCREMENT number of stored elements and write pointer
```

```
162             obj.index = mod(obj.index, obj.len) + 1;
163             obj.stored = min(obj.len, obj.stored + 1);
164         end
165 
166         function idx = top(obj)
167             %TOP Get index of oldest element in queue
168             idx = obj.position(1);
169         end
170 
171         function idx = bottom(obj)
172              %BOTTOM Get index of the youngest element in the queue
173             idx = obj.position(obj.stored);
174             %idx = mod(obj.index + obj.length - 1, obj.length);
175         end
176 
177         function idx = position(obj, n)
178             %POSITION Get actual index for given nth oldest elementob
179             idx = mod(obj.index - obj.stored + n-2, obj.len) + 1;
180         end
181     end
182 end
```

FSOS Test Simulation Script

The MATLAB® script `testSolver.m` simulates the typical run of the FSOS application, with an offline dataset. This allows the testing the implementation on a large period of data as if it were done in realtime, except at a much faster pace.

Listing D.13: testSolver.m (Class)

```
% FSOS - Fine Screen Oversize Solver
% Sensitivity test - Online estimation simulator
%
% Alex Kapor 2021
%

% Load in saved data chunks
filelist = ["datachunk(6)-20210414+30d.mat",...
            "datachunk(5)-20210514+30d.mat",...
            "datachunk(4)-20210613+30d.mat",...
            "datachunk(3)-20210713+30d.mat",...
            "datachunk(2)-20210812+30d.mat",...
            "datachunk(1)-20210911+30d.mat"];

% Make a single matrix
data = nan(3110407,10);
stt = 1;
for f = 1:length(filelist)
    load(filelist(f));
    stp = stt + length(pidata.values)-1;
    data(stt:stp,:) = [pidata.values datenum(pidata.timestamps)];
    stt = stp;
end

%% Perform FSOS simulation
samples = length(data);

% Fix the coefficient bounds
lb = [0.2*ones(1,8) zeros(1,8)];
ub = [0.8*ones(1,8) 0.01*ones(1,8)];

% Window sizes to test
```

```
windowSizes = [1e5 2.5e5 5e5 1e6 1.5e6];
for windowSize = windowSizes
    fprintf("Window Size = %d\n", windowSize);
    % Create aggregator object (Forgetting factor Initially zero - long period model only)
    solver = fsos.cls.Aggregator( 'WindowSize', windowSize, 'NumOfCoefficients', 16, ...
                        'LowerBounds', lb,...
                        'UpperBounds', ub,...
                        'ForgettingFactor', 0.0);
    % Initialise solver
    solver(zeros(1,9));

    % Input validation
    validator = ValidateInput();

    % Record sim results
    updateRate = 10000;
    printRate = 25000;
    coeff = nan(samples,16);
    scrn_recirc = nan(samples,8);
    recirc = nan(samples,1);
    goodcount = 0;
    goodtimes = nan(samples,1);
    for idx = 1:samples
        % Validate inputs
        validated = validator(data(idx,1:9));
        if ~isempty(validated)
            % If good, add to solver
            solver(validated);
            goodcount = goodcount + 1;
            goodtimes(goodcount) = data(idx,10);
        end

        % Record current coefficients, recirc and total oversize.
        coeff(idx,:) = solver.Coefficients;
        scrn_recirc(idx,:) = solver.predictSplit(data(idx,1:8));
        recirc(idx) = solver.predictTotalOs(data(idx,1:8));

        % Print display every 25000 samples
        if mod(idx,printRate) == 0
            fprintf('Sample Time(%d): %s\n', idx, datestr(data(idx,10)));
        end

        % Update model every 10000 samples
```

```
76          if mod(idx,updateRate) == 0
77              solver.update();
78          end
79      end
80      % Collate results for this test run
81      goodtimes(isnan(goodtimes)) = [];
82      filename = sprintf('results-ff-w%d.mat', windowSize);
83      save(filename, 'coeff', 'scrn_recirc', 'recirc', 'goodtimes');
84  end
```

Screen Feed Ratio Optimisation

The MATLAB® script `feedratio.m` is a script that explores how the manipulation of the screen feed ratio impacts the ball mill feed, as discussed in Section 4.4. A series of surface plots are generated to visualise the optimisation space, and demonstrate the optimum solution for a given total feed rate.

Listing D.14: feedratio.m (Script)

```
% feedratio.m
%
% Alex Kapor 31/09/2021
% This script analyses the impact of modify the screen feed ratio on mill
% feed.
%
% The coefficents of a1, b1, a2 and b2 were taken from the oversize solver,
% specically for screens 1A and 1B.
%
clc; clear; close all;

% Model coefficients
a1 = 0.43;
b1 = 1.815e-5;
a2 = 0.2220;
b2 = 1.574e-4;

% Create model objects for both screens
screenA = model(a1, b1);
screenB = model(a2, b2);

% Constraints
maxScreenFeed = 1750;
maxMillFeed = 1850;

% Anon functions for the oversize mass flow of a pair of screens
oversize = @(feeda, feedb) (screenA.os_feed(feeda) + screenB.os_feed(feedb));
% Undersize is just the total feed subtract the oversize.
undersize = @(feeda, feedb) (feeda+feedb) - (oversize(feeda,feedb));

% Calculate total mill feed for given feed rates to screens A and B
```

```
xy = 0:10:maxScreenFeed;            % Screen A & B Feed (same bounds)
z = zeros(length(xy),length(xy));   % Mill Feed
for p = 1:length(xy)
    for q = 1:length(xy)
        z(q,p) = undersize(xy(p),xy(q));   % Mill feed is undersize ore stream
    end
end

% Remove points that exceed maximum allowed mill tonnes
z(z>maxMillFeed) = NaN;

% Plot as 3D Surface
fig = figure('Name', 'Effect of Screen Feed Rates on Mill Feed', 'Position', [0 0 1100 800]);
surf(xy,xy,z, 'EdgeColor', 'none', 'FaceAlpha', 0.8);
xlabel('Screen A (tph)');
ylabel('Screen B (tph)');
zlabel('Feed to Mill (tph)');
zlim([0 maxMillFeed]);
ylim([0 maxScreenFeed]);
xlim([0 maxScreenFeed]);
hold on;

% Plot optimum screen feed line
% Function maximises mill feed as total feed is increased linearly,
m = b1/b2;
c = (a1-a2)/(2*b2);
xr = 0:10:maxScreenFeed;
yr = m*xr + c;
zr = undersize(xr,yr); % Actual mill feed at these rates
plot3(xr,yr,zr, 'm', 'LineWidth', 1);

% Generate constant total feed lines
feedrates = 500:500:3000;
step = 1;
for rate = feedrates
    % rate = x + y (with bounds applied)
    x1 = max(0,(rate-maxScreenFeed)):step:min(maxScreenFeed, rate);
    y1 = rate-x1;
    z1 = undersize(x1,y1);
    ln = plot3(x1,y1,z1, 'k', 'LineWidth', 1);

    % Show datatips on limited range
    if rate >=1000 && rate <=2500
```

```
75          maxidx = find(z1==max(z1));
76          if length(maxidx) > 1
77              maxidx = floor(median(maxidx));
78          end
79          dtt = ln.DataTipTemplate;
80          dtt.DataTipRows(1).Label = 'Screen A';
81          dtt.DataTipRows(2).Label = 'Screen B';
82          dtt.DataTipRows(3).Label = 'Mill Feed';
83          dtt.DataTipRows(end+1) = dataTipTextRow('A + B',x1+y1);
84          datatip(ln, x1(maxidx), y1(maxidx), z1(maxidx));
85      end
86  end
87
88  exportgraphics(fig, 'ratiocontrol1.png', 'Resolution', 400);
89
90  % Transformed onto Ratio vs Total Feed Axes
91  feed = 0:10:3500;
92  s1ratio = 0:0.01:1;
93  z2 = zeros(length(s1ratio),length(feed));
94  for p = 1:length(feed)
95      for q = 1:length(s1ratio)
96          feeda = feed(p)*s1ratio(q);
97          feedb = feed(p) - feeda;
98          z2(q,p) = undersize(feeda,feedb);
99      end
100 end
101 % Remove points that exceed maximum allowed mill tonnes
102 z2(z2>maxMillFeed) = NaN;
103
104 % Plot as 3D Surface
105 fig = figure('Name', 'Effect of Feed Ratio on Mill Feed', 'Position', [0 0 1100 800]);
106 surf(feed,s1ratio,z2, 'EdgeColor', 'none', 'FaceAlpha', 0.8);
107 xlabel('Total Feed, M_{t} (tph)');
108 ylabel('Screen A Ratio, r');
109 zlabel('Feed to Mill, M_{u}(tph)');
110 zlim([0 maxMillFeed]);
111 ylim([0 1]);
112 xlim([0 2*maxScreenFeed]);
113
114 % Plot optimum screen feed ratio
115 % Function maximises mill feed as total feed is increased.
116 r = (feed - c)./(feed+feed*m);
117 z3 = undersize(feed.*r, feed.*(1-r));
```

```
118 hold on;
119 ln = plot3(feed,r,z3, 'm', 'LineWidth', 1);
120 loc = floor(length(feed)/3);
121 tex = ['\fontsize{14}{0} $$r_o=\frac{M_t-(\alpha_1-\alpha_2)/2\beta_2}'...
122       '{M_t(1 + \beta_{1}/\beta_{2})}$$'];
123 text(feed(loc), r(loc), z(loc)+200, tex, ...
124             'HorizontalAlignment', 'right',...
125             'Interpreter', 'latex');
126
127 exportgraphics(fig, 'ratiocontrol2.png', 'Resolution', 400);
```

Simple Quadratic Oversize Model Class

The MATLAB® class `model.m` is a very basic class used to encapsulate the basic quadratic oversize model, its coefficients and functions to calculate oversize ratio and oversize mass flow from a given feed.

Listing D.15: model.m (Class)

```
% model.m
%
% Alex Kapor 31/09/2021 - This simple class encapsulates the coefficients of a
% quadratic oversize model and provides functions to calculate either oversize
% ratio or oversize mass flow from a given feed rate.

classdef model
    % MODEL Basic quadratic model object
    properties
        a % Alpha
        b % Beta
    end
    methods
        function obj = model(a_,b_)
            %MODEL Construct an instance of this class
            obj.a = a_;
            obj.b = b_;
        end
        function out = os_ratio(obj,feed)
            %OS_RATIO = alpha + beta*feed;
            out = obj.a + obj.b.*feed;
        end
        function out = os_feed(obj,feed)
            %OS_FEED = os_ratio*feed
            out = obj.os_ratio(feed).*feed;
        end
    end
end
```

Appendix E

Graphical User Interface

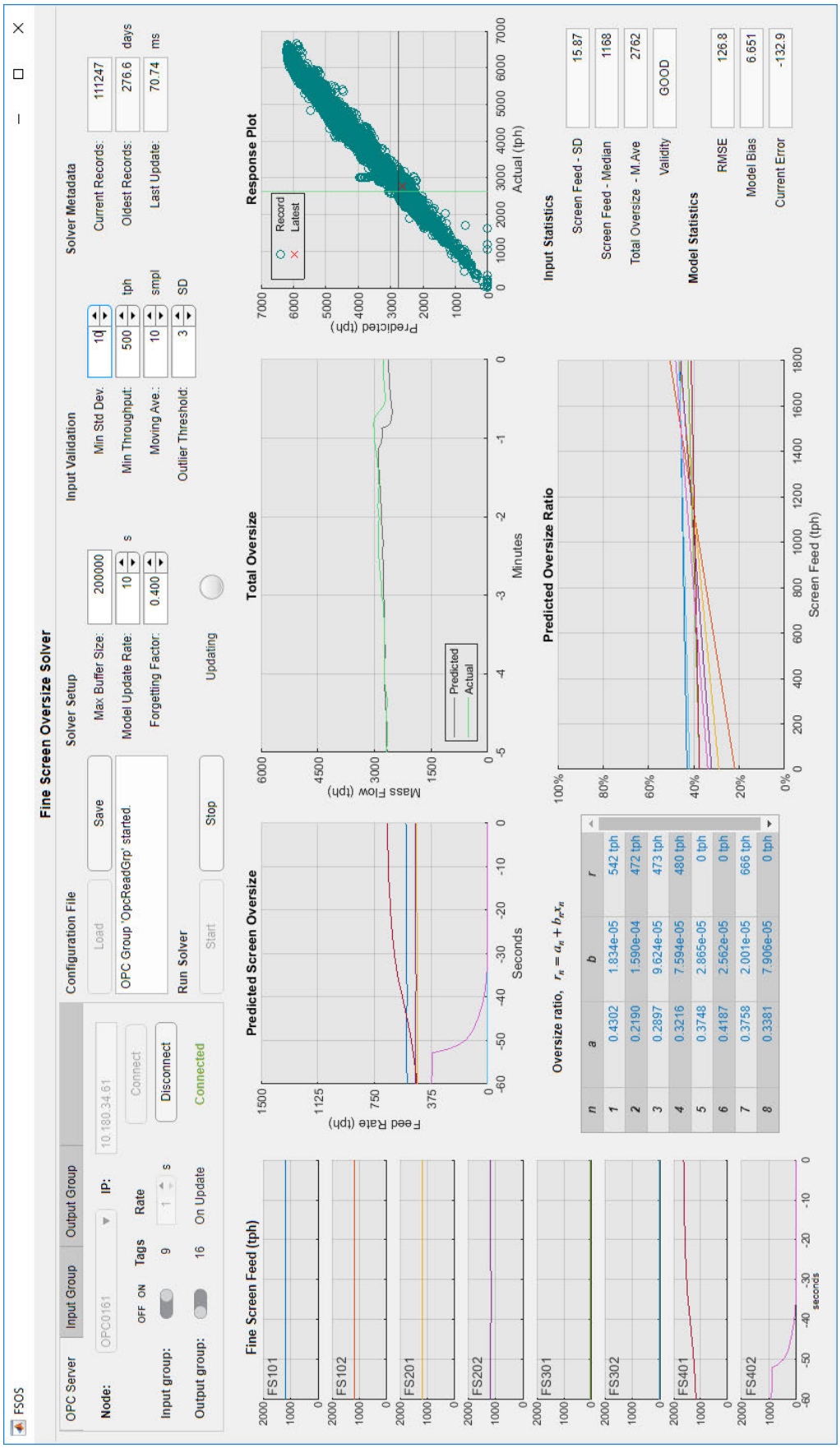

Figure E.1: Screenshot of GUI developed for this project.