Adaptive Model Predictive Control with Deep Models

by
Riley Ballachay

Supervised by

Dr. Bhushan Gopaluni

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE COURSE

CHBE 496: CHEMICAL AND BIOLOGICAL ENGINEERING THESIS, BIOTECHNOLOGY TOPIC

in

The Faculty of Applied Science
(Chemical and Biological Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

April 2021

Adaptive Model Predictive Control with Deep Models

Riley Ballachay[1], Bhushan Gopaluni[1]

Department of Chemical and Biological Engineering, UBC, Vancouver, Canada

Abstract

We present a machine learning (ML) framework to identify linear single-input, single-output (SISO) and multi-input, multi-output (MIMO) systems from simulated sets of input/output data. More specifically, we demonstrate that recurrent neural networks (RNNs) offer a rapid and simplified method of estimating first order, discrete-time, transfer function parameters with associated uncertainties from short observations of system dynamics. Simulations are produced by generating linear responses to random sequences of step inputs with randomly selected combinations of system parameters. We demonstrate that system parameters can be identified quickly from short sequences of data, so that the model can be integrated into on-line model predictive control (MPC) control systems.

Email address: riley.ballachay@gmail.com (Riley Ballachay)

Contents

Nomenclature

α_s	Closed-loop response speed
β_s	Dominant settling time
$\hat{y}_i$	Auto-encoder output prediction
λ	Noise variance
ω	Signal frequency
ϕ	Neural network activation function
τ	FOPDT time constant
τ_{dom}	Dominant system time constant
θ	Model coefficient vector
θ_D	FOPDT dead time
ANN	Artificial neural network
ARX	Auto-regressive with exogenous input
b	Neural network bias
C_t	Recurrent neural network memory update function
CNN	Convolutional neural network
CV	Controlled variable
D	Auto-encoder training set
D_y	Signal delay
DT	Discrete-time system model
$ELBO$	(Evidence Lower BOund) objective
f_t	Recurrent neural network forget gate
$FOPDT$	First order plus dead time transfer function
H	Bayesian hypothesis

h_S	Recurrent neural network hidden state
i_t	Recurrent neural network input gate
J	Quadratic cost function
K	FOPDT process gain
k	Model time step
KL	Kullback-Leibler divergence
LS	Least-squares method
$LSTM$	Long-short-term memory network
$MIMO$	Multi-input, multi-output system
MPC	Model predictive control
MV	Manipulated variable
n_d	Number of output signals
n_r	Number of binary signal registers
N_s	Number of signal periods in a window
n_u	Number of input signals
o_t	Recurrent neural network output gate
P	Probability distribution
p	Probability
PID	Proportional, integral, derivative controller
$PRBS$	Pseudo-random binary signal
Q	Tractable probability distribution
RNN	Recurrent neural network
SID	System identification
T_{settle}^{max}	Maximum system settling time

T_w	Signal clock period
T_{max}	Maximum identification time
TF	Transfer function
u	Input/manipulated system variable
v	Sensor noise variable
w	System disturbance variable
W_j	Neural network tensor weights
X	Input array
x	System state variable
Y	Output array
y	Output/controlled system variable
Z	Latent representation
z	Model predictive control horizon

1. Introduction

1.1. Introduction

System identification (SID) is a discipline of control engineering concerned with representing systems as mathematical models which can be used to predict or explain observed behaviour [1]. SID is indispensable in modern control, where a system representation is frequently used to better understand behaviour and minimize controller error [2]. In the most common form of process control: proportional integral derivative (PID), SID is used to tune controllers and minimize set point tracking error. One form of control to which an accurate system model is central is model-predictive control (MPC). MPC has become extremely popular in high dimension, multi-input, multi-output (MIMO) systems with complex mechanics owing to low set point error and capacity to handle variable constraints. It has been a particularly popular control strategy in the petroleum and chemical industries since the 1980s [3]. Control is achieved through solution of the optimal control action to take over a finite time horizon. The first control action is realized and the control problem is solved again, resulting in recursive, closed-loop control. It has been determined that discrete-time (DT), linear, empirical models are the optimal representation for this method of control [4]. Many different models falling into this class have been implemented in MPC. The simplest of these models is auto-regressive with exogenous inputs (ARX). This model represents observed outputs as a linear sum of past inputs and outputs.

Traditional SID techniques for MPC are extremely time-consuming and expensive. Many control engineers consider it to be the most intensive part of implementing MPC systems [5]. The predominant identification method has historically been single-variable testing - either *step* or *impulse response*. In these identification procedures, the system is held at steady state and each manipulated variable (MV) is manually delivered a *step* or *impulse*. The resultant trajectory of every controlled variable is recorded until the system returns to steady state. Linear regression is then used to parameterize the model for each controlled variable. For a crude distillation column with 34 MVs and 94 CVs, single-variable identification can take up to about 30 days. A skilled operator with enough knowledge of the system under study can test multiple MVs concurrently to reduce testing time. Significant research has been conducted into methods for combining variables to expedite this process. Unfortunately, no matter how fast and accurately open-loop SID tests can be conducted, systems models will inevitably become defective

and must be re-trained. As the process is run, problems arise, valves and pumps fail, exchangers and pipes foul and sensors become distorted, leading to significant deprecation in model accuracy and controller performance. It is harder to justify halting operation of an aging process to re-identify an increasingly erratic system model.

There has been considerable progress in identification algorithms to decrease the complexity and duration of SID. The most significant of these developments is the evolution of SID algorithms to train on closed-loop data. This has enabled the deployment of SID in online MPC, where drifting parameters are recognized and updated in real time [6]. One software package which has seen particular success in addressing practical SID problems is Tai-Ji ID Automatic Closed-Loop Identification Package for Model Based Process Control [7]. It configures multi-input, multi-output (MIMO) identification tests, interprets the system response and validates the model independently [8]. A newer version of Tai-Ji, Tai-Ji Online, allows some CVs to be under closed-loop control while the identification procedure is being conducted. Model identification is scheduled for every few hours, so that fresh models are obtained at desired intervals [9]. Among the companies who use Tai-Ji ID software are Dow Chemical (Netherlands, Germany, USA, Canada), British Petroleum (Netherlands, UK, USA) and Exxon-Mobil (Globally).

The critical advantages of automated SID packages like Tai-Ji include thoughtful multivariable system testing, semi-closed-loop control and scheduled testing. While automated SID methods leverage a few advanced techniques, including simultaneous MV and closed-loop testing, the majority still rely upon relatively basic least-squares (LS), maximum-likelihood or prediction-error methods (PEMs), which have been in use since the first implementation of MPC. Reviews of parameter estimation methods have revealed that LS methods are computationally expensive and result in up to 25% parameter error in some cases [10]. As industrial systems become more complex and the requirements for controller precision become more stringent, the demand for algorithms capable of accurately solving complex problems with low computational overhead intensifies. More advanced algorithms exist and have had substantial success in parameterizing DT models, however none has addressed the practical limitations of an industrial SID package. Artificial neural networks (ANN), which have seen extensive real-world application in fields including image recognition, system identification and control, have been of particular interest for SI. This owes to their abil-

ity to successfully approximate many complex nonlinear relationships [11]. The main concern with ANNs, historically, has been the capacity to find any rationale or physical justification for their behaviour [12]. The merit of this concern is significantly diminished when using empirical linear models, however, as success is gauged by prediction accuracy alone.

Many ANN architectures have been applied to SID problems. Among these architectures, recurrent neural networks (RNNs) are one the most attractive. RNNs are a variation of ANN developed specifically for learning dynamical or time-dependent relationships between inputs and outputs. RNNs have been used to great success in parameterizing DT linear models. A 'Linerec' neural network, a novel RNN architecture by Fei et al (2006), was used to successfully parameterize high-order DT TFs from pulse response data [13]. The system input was fed into the network, and the network was trained to predict the output signal. Linear combinations of the final layers of tensors was then used to parameterize the TF. Convolutional neural networks (CNNs), a variation of ANN developed for 2D/3D data like images, have also shown promise in SID tasks. Genc (2017) used convolutional filters to extract parametric information from input and output matrices and successfully used the weights to parameterize a Nonlinear Auto-regressive with Exogenous Input Model [14].

While the methods developed to date have demonstrated that linear temporal relationships can be extracted using neural networks they fail to address a few practical considerations. The first and most significant practical limitation of these methods is the computational intensity. Fei et al noted that the accuracy of the recurrent 'Linerec' network is reliant upon significant 'training numbers' and more than 1000 epochs for proper convergence. Genc reported that solution of a 5x3 system with their CNN required 36 minutes of system data, 50 filters, 5000 epochs and 400 batches to achieve achieve target accuracy on a training set. The computational appetite of neural network training is well-known and well reported in the literature [15]. Re-optimizing such models online poses a significant hardware and software requirements for controllers. One of essential parts of an online SID package is the ability to re-parameterize frequently. Requiring training data that spans hours of operation may not capture shifting dynamics in a desirable way.

While ANN's have shown promise in parameterizing linear empirical models for SID, they have significant practical limitations. An ideal SID

method would be capable of parameterizing large MIMO systems quickly and with low computational overhead. The alloy of these capabilities would allow a lean, intelligent network to be deployed online in closed-loop. The proposed method for achieving these goals will incorporate the proven capabilities of RNNs and CNNs, while adding the ability of the network to be trained offline prior to deployment in closed-loop control. Additional to allowing the network to be trained offline prior to deployment, it is desired that the parameters are produced with some estimate of uncertainty, so that new parameters may be rejected in outlying cases.

1.2. Motivating Examples

Distillation columns are an extremely sensitive unit operation in the petroleum refining process, owing to large economic incentives for small improvements in performance [16]. The optimal operating conditions of the column fluctuate in real-time based on market conditions and the control system must counteract system disturbances to maintain product composition within fixed constraints. The control system of the distillation column is thus tasked with achieving constrained multivariable optimization online, making it well-suited for MPC.

When a theoretical model isn't available, as is usually the case for crude distillation columns, an empirical model is often used to develop a model-based controller [17]. Data for parameterizing these models is generated through a series of single variable step tests during a process known as system identification. These tests are carried out manually by an experienced operator, who uses a combination of intuition and protocols to ascertain process behaviour [18]. The duration of a single step can be up to 10000 minutes for some distillation columns, at a sampling interval of 2 minutes for a total of 5000 samples [19]. Manipulated variables for a typical crude column include feed flow rates to each tray, side draw rates and reflux rates. Critical controlled variables include level control in side strippers, product flashpoint, furnace duty, total crude feed rate and valve positions [18]. For a crude distillation unit with approximately 20 MV's and 40 controlled variables CV's, identification can take anywhere from two to four weeks.

Once the complete data set is obtained, operators can proceed to identify the transfer functions defining the multivariable system and validating the model during a typical session of operation. There are extensive problems with this common method of identification reported in the literature. The biggest problem is the high cost of manpower and down-time to fully

identify the process. Additionally, single-variable tests typically result in models with unsatisfactory control performance and information on multivariable system characteristics under closed-loop control [19]. Some efforts have been made to improve identification for MPC in process control. One verified approach is to use pseudo-random binary sequence (PRBS) as a single-variable input to reduce disturbances to product quality and improve data resolution [20]. Distinct from the single-variable testing which traditionally uses non-parametric step or impulse response modeling [21], this identification process requires use of the asymptotic method (ASYM) of identification [22]. Multivariable testing has also been used to relative success with both correlated and uncorrelated PRBS signals [19]. Producing input data as a stationary process and testing multiple MVs at once is the most critical step in developing closed-loop identification in order to address the aforementioned problems, as demonstrated by Zhu et al (1998).

The asymptotic method is one of the most widely used algorithms in the prediction error model family. Prediction error methods have the benefit of being applicable in closed loop and providing the best possible results (minimum covariance) provided the true system is contained within the model space. The critical drawback of these algorithms is computational intensity, sensitivity to initial parameter estimates and lower efficiency in parameterizing nonlinear models with non-convex optimization spaces [23]. Neural-network (NN) models have been of particular interest in distillation column control, owing to their ability to handle highly complex and nonlinear interactions. NN models have be used to great success as a model capable of predicting crude distillation column behaviour [24], and have been implemented directly as the internal model in MPC of a crude column [25].

2. Background

2.1. Systems Theory

From hereon, we will consider a system to be an object containing variables ranging in scale which interact according to first-principles laws and produce an observable effect [17]. System behaviour is perceptible to an observer through a sensor. The sensor is an integral component of the system, subject to its own dynamics and stochasticity. The basic graphical representation of a system is shown in Figure 1. This figure contains the essential system variables plus critical information flow.

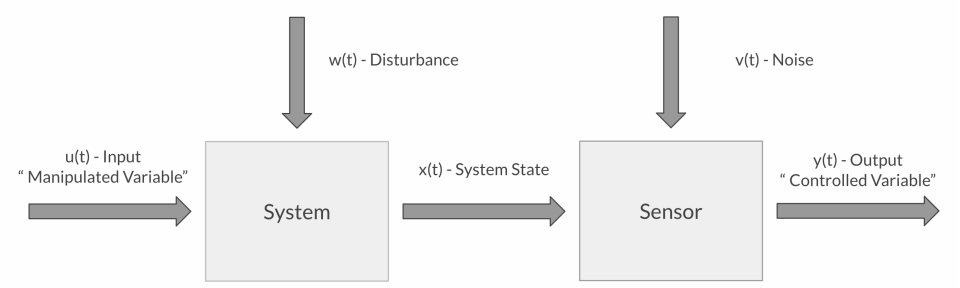


Figure 1: Simplified diagram of a dynamic system

Input, u: The input, u, is measurable and manipulable by the observer. It acts in concert with disturbances to produce system behaviour. Will also be referred to as the "manipulated" variable.

Disturbance, w: The disturbance, w, can be measured in some cases, but is not manipulable by the observer. It can have varying degrees of impact on system behaviour.

System state, x: The system state, x, is the sum of variables which describes the system at a given point of time. Results from interaction of inputs and disturbances.

Sensor noise, v: The sensor noise, v, is structured or unstructured noise produced by interpretation of the system state by the sensor.

Output, y: The output, y, is the interpretation of the system state by the sensor, subject to bias and noise v. For purposes of control, y will also be referred to as the "controlled" variable.

Control refers to calculation of an input u(t) which achieves a desired trajectory for the output variable y(t). Control exists in two essential flavours. The first is open-loop control, in which the input, plus a system model, is used to predict the behaviour of the controlled variable. The controller then manipulates the input to reduce the error in the output. Integral to this control strategy is an accurate model of the system. Closed-loop control uses the trajectory of the output to inform the trajectory of the input, creating what is referred to as 'feedback' control.

2.2. Auto-regressive Plus Exogenous Input

ARX models may be represented in their simplest form as a linear difference equation of inputs and outputs [26]:

$$y(t) + a_1 y(t-1) + ... + a_n y(t-n) = b_1 u(t-1) + ... + b_n u(t-k) + e(t) \quad (1)$$

Where y is the system input, or controlled variable, u is the output, or manipulated variable, e is added noise and k is the backwards time-shift operator. a_n and b_n are coefficients of the polynomial relating each past time-point to the current output. The first-order form of the difference equation contains two coefficients and the time delay parameter n:

$$y(t) = bu(t-k) - ay(t-1) + e(t) \quad (2)$$

Once in this form, the prediction model may also be represented as a discrete-time transfer function. A certain amount of system and sensor noise will be coupled with the true system behaviour in the final observed signal. An ordinary ARX model considers disturbance added to the input channel as well as output. For the purposes of modelling the true underlying linear model, we consider the input to be deterministic and noise to be added at the sensor measurement. This slight variation on the ARX model is known as an output-error (OE) model. This model contains three parameters relating inputs and outputs, with an additional term for random noise (e(t)) [27]:

$$G(t) = \frac{bq^{-k}}{1 + aq^{-1}} + e(t) \quad (3)$$

This transfer function relates each pair of inputs and outputs in a system. For a system with multiple inputs and outputs:

$$\begin{pmatrix} G_{1,1}(t)\, G_{1,2}(t)\, G_{1,n}(t) \\ G_{2,1}(t)\, G_{2,2}(t)\, G_{2,n}(t) \\ G_{n,1}(t)\, G_{n,2}(t)\, G_{n,n}(t) \end{pmatrix} \times \begin{pmatrix} u_1(t) \\ u_2(t) \\ u_n(t) \end{pmatrix} = \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_n(t) \end{pmatrix}$$

For each set of system inputs $u(t) \in R^m$ and outputs $y(t) \in R^n$, there exists an underlying transfer function matrix $G(t) \in R^{m \times n}$, containing the process parameters a, b and k. Linear combinations of system inputs u(t) and y(t), will be used to infer parameters in before unseen transfer matrices with parameters within the parameter space of training data.

2.3. System Identification

The general procedure for identifying a system varies depending upon prior knowledge and complexity of the system under study. All system models can be placed into three general categories:

White-box: Derived from first-principles laws, requires proper observation of essential state variables, are developed prior to data collection and parameterized using collected data.

Black-box: Don't necessarily relate to any underlying physical relationships. Frequently use soft-sensoring to estimate underlying state variables and are built and parameterized after data collection.

Gray-box: Use a combination of white- and black-box techniques. Typically used in cases where state variables are partially inaccessible or physical modeling is inadequate.

Discrete-time linear models, as used in MPC, are most commonly identified using discrete impulse identification. In this procedure, a system is held at steady-state under open-loop control by an operator using prior knowledge of system dynamics. An impulse is introduced into each manipulated variable and the complete response of each controlled variable is recorded. Once the system returns to steady-state, the next manipulated variable can be perturbed. This process is extremely time-intensive and requires an experienced operator to ensure the system remains stable. For a crude distillation column with 34 manipulated variables and 120 controlled variables, the entire identification process can take up to 30 days. Once the full set of dynamic data is recorded, linear regression is used to obtain coefficient vectors A,B.

2.4. Model Predictive Control

Optimal control is an indispensable modern development in the field of systems theory. Generally, optimal control is a technique of minimizing a cost function representing system performance through choice of a set of input trajectories. The most successful form of optimal control, Model Predictive Control (MPC), is used widely in cases of linear state feedback. The general goal of MPC is to predict the trajectory of the output variable y(t) given the current state and possible control actions in u(t). A discrete control action is taken, the cost is computed and the optimization problem is

solved again. Through repeated computation of the optimal control strategy, the controller error is gradually reduced and feedback control is achieved.

The auto-regressive plus exogenous (ARX) model utilized in MPC can be represented as follows:

$$y(t) = Ay_k + Bu_k \tag{4}$$

Where y is the system output, assumed to be linearly related to the system state, y_k is a vector of system outputs, u_k is a vector of system inputs and A/B are coefficient vectors. The control problem is formulated as an optimization problem seeking to drive the system state to a reference point with bounded acceptable control action. If we assume that the reference state is set at the origin, and control action is taken directly on the system state x_k, we get a quadratic cost function of:

$$J(x_0, u_0, u1, ...u_k) = \sum_{k=0}^{\infty} (\|x_k\|_Q^2 + \|u_k\|_R^2) \tag{5}$$

Where Q and R are weighting matrices that emphasize state and input significance to optimal control. At each time t, the optimal control action minimizes the control sequence over a control horizon z denoted by:

$$J_{min} = min\ J(x_k, [u_t, u_{t+1}, ..., u_{t+z}]) \tag{6}$$

The problem of maintaining a reference state in the controlled variable is addressed by redefining the system state in terms of the optimal steady-state value.

2.5. Project Approach to Model Predictive Control

While the preceding discussion has primarily concerned itself with parameterizing an existing process model, process identification projects require significant advance preparation and subsequent validation prior to model deployment. This section intends to elucidate the context of parameterization in industrial control, to convey a more complete picture of system identification. The general project approach outlined by Zhu (2006) can be broken down into six main steps:

1. Benefit Analysis and Functional Design
    - Study the process, identify constraints and set control objectives

- Preliminary MPC with control goal is outlined

2. Pre-test
   - Identify and fix instruments, inspect PID loops
   - Estimate process time, time delays and dominant gains
3. Identification Test and Model Identification
   - Identification experiment
4. Controller Tuning and Simulation
   - MPC is tuned according to control goals
   - Priorities are weighted to solve control conflicts
5. Controller Commissioning
   - Controller is first tested in off-line mode
   - Once validated off-line, model can be deployed
6. Controller Maintenance
   - Fouling will necessitate re-identification
   - Closed-loop model re-parameterization

While discussing the complete identification process at length is beyond the scope of this project, it is important to acknowledge that many simplifying assumptions have been made. One principal assumption is that the underlying process model used in this project is linear, continuous-time and parametric. While this enables the synthesis of artificial data for training and testing the parameterization model, it is not a perfect representation of a real process. For a more exhaustive discussion of the complete identification procedure see Isermann (1979), Liu et al (2012) and Gao et al (2019).

2.6. Deep Learning

Deep learning is a subsection of machine learning that uses artificial neural networks (ANN) to simulate cognition. ANNs map feature variables X to a target variable Y through nonlinear interactions [28]. Each network consist of a variable number of perceptrons stacked into layers and connected by neurons. Each neuron in has a weight w_j, and can either be active or inactive. Perceptrons sum all neurons in j, add bias, b, and pass the sum to an activation function, ϕ

$$a = \phi(\sum_j w_j x_j + b) \tag{7}$$

If a threshold is reached, the perceptron activates neurons in thee subsequent layer which are connected to it.

2.7. Recurrent Neural Networks

Recurrent neural networks are a variation of conventional neural networks which handles variable-length sequences by introducing hidden states which store information from previous time states [29]. Given a sequence of input data x = (x_1,x_2,...x_t), the RNN updates the hidden state h_S by

$$h = \begin{cases} 0, & t = 0 \\ \phi(h_{t-1}, x_t), & otherwise \end{cases} \tag{8}$$

where ϕ is a logistic activation function applied element-wise on the input vectors. Typically, the input vectors are linearly transformed and summed before applying the activation function. As in feed-forward neural networks, error is propagated backwards through the network during back-propagation using the chain rule. As error gradients are transferred backwards through the network, they can shrink or grow exponentially, leading to either the vanishing or exploding gradient problem [30]. Long/short term memory networks (LSTMs) are a variation of RNNs created to address the vanishing gradient problem [31]. LSTMs store memory as a cell state vector, which is regulated by three sigmoid-activated gates and a memory update function: a forget gate (f_t), an input gate (i_t), an output gate (o_t) and a memory update function (C_t). The input gate regulates information from the prior cell, the memory function aggregates old and new memory, the forget gate controls state persistence within the cell and the output gate regulates the information transfer to subsequent cells. In compact form, the more complex form of the LSTM cell can be written as

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{9a}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{9b}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{9c}$$
$$\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c) \tag{9d}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{9e}$$
$$h_t = o_t \odot \sigma_h(c_t) \tag{9f}$$

2.8. Variational Auto-encoding

Effectively estimating and managing uncertainty in predicted model parameters is essential to maintaining optimal control. Machine learning has

been decreasing in relevance for practical applications with more stringent guidelines for safety and consistency [32]. Engineering is a tightly regulated field and new technology must be grounded in first-principles science and math to gain wide adoption in the field.

A large methodological development in ML has been advances in incorporating uncertainty into deep learning methods. More specifically, two different types of uncertainty: aleatoric, the irreducible stochasticity of real processes and measurements and epistemic, the reducible lack of knowledge about system state and behaviour [33]. In supervised learning, where a model is provided a set of training data $D = (x_1, y_1), ..., (x_N, y_N)$, we seek a hypothesis H from the hypothesis space which adequately captures the variance of the dataset. While most commonly reported, average accuracy of a hypothesis training set D provides no insight to the accuracy of individual queries of the model. Given an individual instance $(x_i, \dot{)}$, it is desired to provide an estimate y_i with a transductive approximation of the uncertainty bounds. As the relationship between input instance x_i and prediction $\hat{y_i}$ is achieved through many approximations propagated by network connections, the reformulation of connections as probability distribution is a more appropriate representation of the relationship [34]. Given a nonlinear encoding of the input sequence $X = [x_1, x_2, ..., x_t]$ of a GRU layer and subsequent dense layer to a latent representation $Z = [z_1, z_2, ..., z_i]$ of reduced dimensionality, the target variables Y and the Bayesian parameters $\theta = [\theta_1, ..., \theta_n]$, we can consider the general problem of deriving a posterior distribution for the hidden layer θ as

$$p(\theta, H|Z) = \frac{p(Z|\theta, H)p(\theta, H)}{p(Z)} \tag{10}$$

where $p(\theta, H)$ is the prior distribution of weights for hypothesis H, $p(Z|\theta, H)$ is the likelihood function and $p(Z)$ is the probability of the latent variables. As the prior $p(\theta, H)$ and posterior $p(\theta, H|Z)$ distributions are intractable, so variational inference is used to approximate the underlying distribution [35]. The need for time-intensive sampling-based methods is circumvented by solving an optimization problem over a class of tractable distributions Q in order to find a $q \in Q$ that approximates the true distribution p. The similarity of distributions q and p is measured by the Kullback-Leibler (KL) divergence [36, 37]. KL divergence isn't directly soluble, so the objective is reformulated as the ELBO (Evidence Lower BOund) objective [38]

$$ELBO(\theta) = -\int dwQ(w;\theta)logP(Y|Z,) + \int dwQ(w;\theta)log\frac{Q(w;\theta)}{P(w)} \quad (11)$$

where $P(w)$ is the prior over the weights, $Q(w;\theta)$ is the variational posterior approximating the true posterior, $P(Y|X,w)$ is the likelihood function relating the latent representation Z, the output to the lambda function Y and the layer weights w.

3. Data Generation for Network Training

Pseudo-random-binary-sequences are a two-state signal generated by combining a deterministic input polynomial and a shifting feedback register [39]. Although the process is theoretically deterministic, the register will generate a signal with an auto-correlation function resembling that of white noise [7].

PRBS signals are generated as input sequences for training and validation data. Accurate identification of complex systems necessitates detailed and thoughtful identification experiments. Not all processes may be excited in similar fashions, as careful consideration must be taken to the frequency band of excitation and cross-correlation between inputs. Rivera and Jun have proposed a methodology reliant upon careful consideration of process dynamics [40]. The following guideline serves as an estimate for the frequency band of input excitation:

$$\frac{1}{\beta_s \tau_{dom}^H} \leq \omega \leq \frac{\alpha_s}{\tau_{dom}^L} \quad (12)$$

where τ_{dom}^H and τ_{dom}^L are estimates of high and low estimates of dominant time constants, β_s is an integer representation of the process settling factor (e.g. for $T_{95\%}$, use $\beta_s = 3$; for $T_{99\%}$ use $\beta_s = 5$) while α_s is the closed-loop response speed, a multiple of the open-loop response time. A PRBS signal has two essential parameters, n_r, the number of registers in a PRBS window and T_w, the clock period. The signal repeats itself after $N_s T_w$ seconds, where $N_s = 2^{n_r} - 1$, using the expression from 9, we obtain the following rules for PRBS parameters:

$$T_w \leq \frac{278\tau_{dom}^H}{\alpha_s} \quad (13a)$$

$$N_s = 2^{n_r} - 1 \leq \frac{2\pi\beta_s\tau_{dom}^H}{T_w} \quad (13b)$$

n_T and N_s are integers and T_w is an integer multiple of the sample time. As mentioned earlier, it is essential that opportunities for cross-correlation in system responses are minimized. The following rule for multivariable signals can be implemented in cases where each input is delivered a PRBS in sequence, shifted by the following delay parameter, D_y:

$$D_y = \frac{T_{settle}^{max}}{T_w} \tag{14}$$

where T_{settle}^{max} is the universal maximum settling time. Each manipulated variable in U^p may then be delivered an identical PRBS sequence, delayed by $(p-1)D$. This method is known as sequential testing. Conner and Seborg [41] performed a comprehensive analysis of three different modes of PRBS test design: sequential, simultaneous and rotated testing. They determined simultaneous input (simultaneous, unstructured PRBS) resulted in the lowest ARX model error for a 2x2 system. In each scenario the authors determined that the following heuristic may be used to determine data length:

$$T_{test} = 6T_{max}(n_u + n_d) \tag{15}$$

where n_u and n_d are the number of inputs and outputs, respectively, and T_{max} is the maximum system settling time. To generate each random system responses, parameters a and b are randomly selected from the continuous range [0.01,0.99], θ is selected from the discrete range [1,10] and used to formulate a new transfer function $G(s)$. A system response to input $u(t)$ is created using the randomly generated PRBS input and randomly parameterized transfer function $G(s)$. Five percent Gaussian noise is added and the output signal $y(t)$ is paired with the input signal $u(t)$ for training. 100,000 system input/response pairs each are generated to train the model. The identification sequence design for a 4x4 MIMO system is shown in Figure 2.

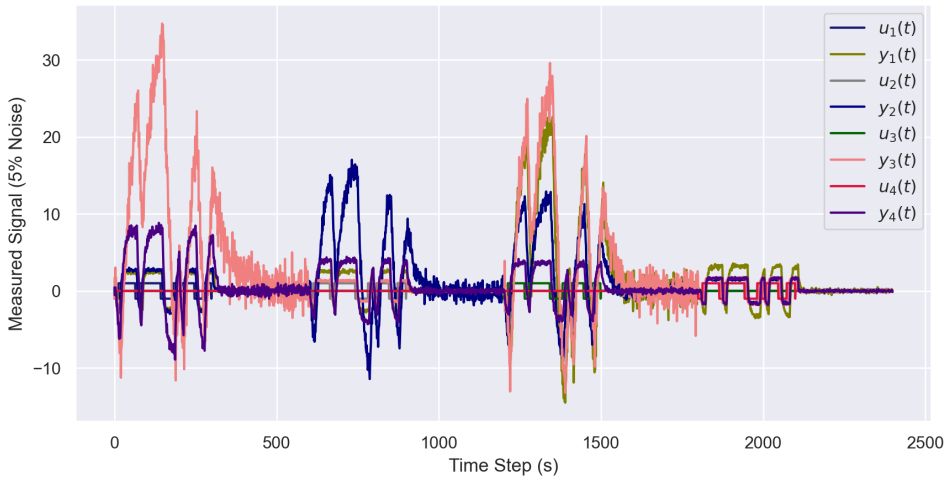


Figure 2: 4x4 MIMO testing sequence simulated in Python

Input set $u(t)|y(t)$ are linearly combined prior to feeding into the network as follows:

$$z(t) = y(t) - u(t) \tag{16}$$

The final 300 seconds of each excited input sequence is dropped from each linear combination and the combinations are stacked to be fed into the first order 1x1 ordinary network. The linear combination of input 1 and outputs 1-4 is shown in Figure 3

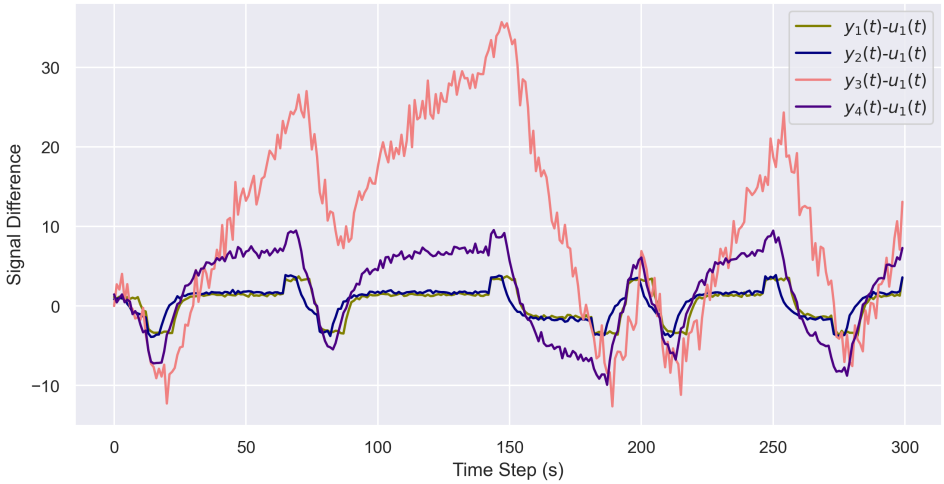


Figure 3: Linear combination of first input sequence and four outputs

3.1. Alternate Methods for Data Generation

PRBS signals address the primary concern of data generation for system identification - persistently exciting with zero mean. Control of a crude distillation column is an extremely complex problem, owing to many stochastic perturbations including feed composition, external temperature and complex dynamics contained within the column [42]. As a result of these stochastic disturbances, the typical MPC action on the manipulated variables of the column end up looking a lot like a random process with small discrete steps and a moderate amount of memory. To test the algorithm designed for use on PRBS input/output signals, we verify on a random process with period similar to the PRBS signal. Figure 4 shows both signals and the corresponding spectra.

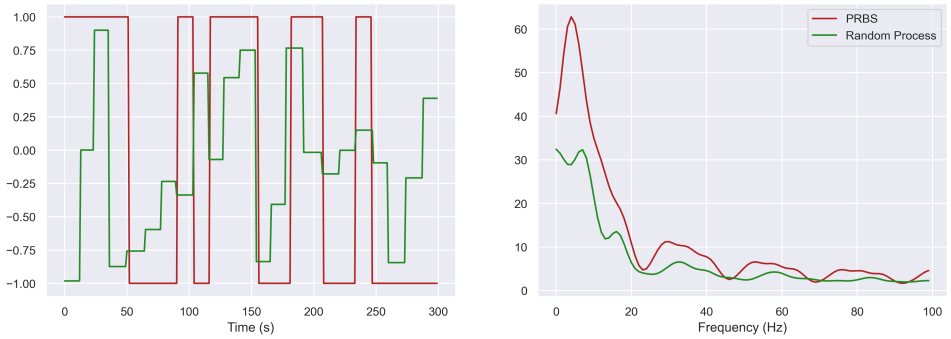


Figure 4: Random process and PRBS data with corresponding spectra

3.2. Optimal Network Configuration

Recurrent neural networks with latent random hidden state variables have demonstrated efficacy on sequential data including natural speech [43] and written text [44]. The VAEs of these model are conditioned on the hidden state variable of the RNN. Therefore, the approximate posterior will not only be a function of the input sequence x_t, but also the hidden state h_t. Other models have used two linear models at each time-step in the RNN to estimate the mean and variance of the latent variable [45]. A linear combination of all the hidden state distributions then may be combined for a single estimate of the desired parameter as a Gaussian distribution. Multiple architectures were investigated to produce a final network with strong inference potential and meaningful estimates of parameter uncertainty. The best model investigated in this paper utilized an LSTM architecture with VAEs over the LSTM latent representation of each input sequence. The final combination of the latent variables produces an estimate of each parameter a, b and k as a Gaussian distribution, for a total of 3 models for each first-order transfer function.

All models were implemented using Tensorflow and Tensorflow Probability [46]. RNN models of multiple dimensions were investigated, but the best optimal RNN length was determined to be 100 nodes single layer LSTM. The output activation function from each LSTM cell was reformulated as a Gaussian prior distribution with a mean of the tensor weight and standard deviation equal to 0.1. The Gaussian tensors are pooled into two linear nodes with KL weight as the inverse of batch size. The posteriors from the two linear nodes represent the variance and mean of the final parameter estimate, modelled as normal distributions. Each posterior distribution has a mean of the tensor weight and variance proportional to the Softplus trans-

formation of the tensor weight plus $1e-5$. The posteriors are incorporated in a final distribution representing the mean, proportional to the weight of the posterior representing the mean and the variance equal to 0.1 times the the Softplus regularization of the 'variance' posterior plus $1e-3$. Figure 5 shows the architecture of the proposed model.

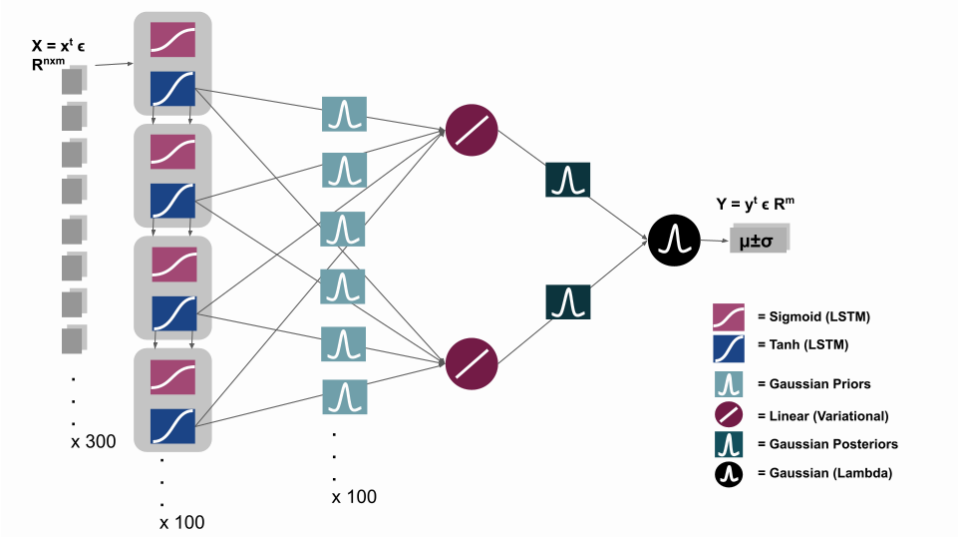


Figure 5: Simplified network architecture developed in Tensorflow

4. Discussion - First Generation Network

This section will discuss the first iteration of research performed on the deep learning model mentioned before, performed between August 2020 and January 2021. Insights made from this initial round of work informed further work completed between February and April 2021. The conclusions made from this work, and the changes made in the second round of work will be discussed.

4.1. System Training

100,000 samples were used to identify and validate all models, with a training/validation split of 70/30. The same network architecture was utilized for identification of a, b and k: 100 units LSTM, 2 units with linear activation and 1 Lambda distribution. All networks were used with default hyper-parameters where not specified. Mini-batch is the most commonly utilized method for stochastic gradient descent in deep networks utilizing

VAEs [47, 48]. A mini-batch size of 32 was determined to lead to optimal convergence and was used for all models. Models were trained using the Adam [49] optimizer with the default learning rate of 0.001. Negative log likelihood was determined to be the optimal loss function for the model, performing better in the majority of probability-maximizing tasks than mean squared error [50].

Figure 6 shows the training curves for the aforementioned network. Models were trained for 500 epochs. A minimum validation loss checkpoint was used, so that model weights leading to the lowest validation loss were stored and saved until the end of training. Training and validation loss curves follow closely, indicating that the validation and training sets are large and similar enough to avoid over-fitting.

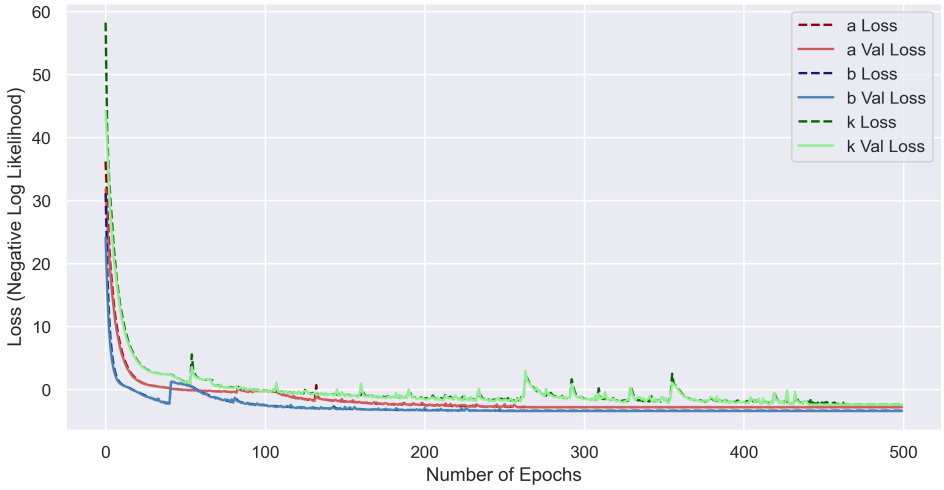


Figure 6: Training and validation loss for a,b and k models

4.2. Test Set Results

Figure 7 shows the results of the universal system model on test sets of 100 samples of unique input/output data, or a 10x10 system tested using sequential experimental data. Testing parameters are all sampled from the same range as training and validation parameters. All validation coefficients of determination are above 0.995. Accuracy and uncertainty bounds on the gain parameter k, are independent of the parameter value. Average uncertainty on ARX parameters a, b and k are 0.01, 0.02 and 0.15, respectively.

Evident from Figure 7 is a variable amount of uncertainty predicted in parameters depending upon the magnitude of the predicted parameter. Ad-

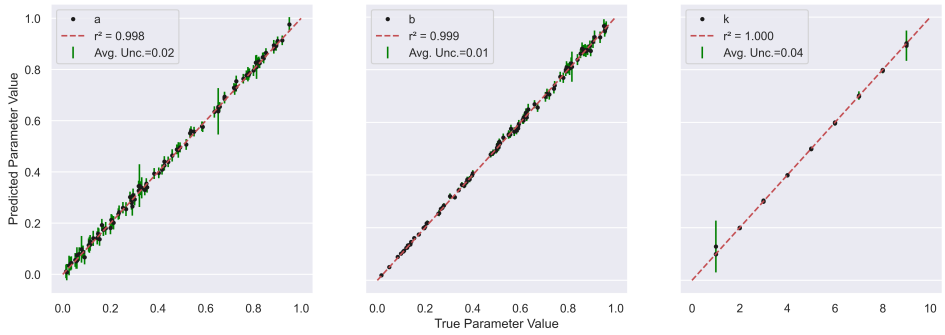


Figure 7: Plot of 100 Gaussian predictions in a, b and k for test parameter range

ditionally, the small uncertainty bounds on parameters b and k evidence that some degree of over-training in the VAE's is occurring. This is possibly a result of the trade-off made during training between optimizing generalization ability of the network by minimizing the size of posterior and prior distributions. Prior research on VAE's have addressed this problem, commonly referred to as the amortization gap by tuning auto-encoder parameters including distribution size and type, thereby increasing the generalization ability [51]. While optimizing this feature of the network is beyond the scope of this research, it is important to note that this limitation has been addressed in prior research through changes to the data and network architecture.

For each multi-input, multi-output system with a matrix of first order transfer functions, each transfer function may be represented as a collection of three distributions, each representing a parameter in [a,b,k], plus the estimated uncertainty in each parameter. For a 4x4 system with parameters in the training and validation space, a transfer function matrix with 48 distributions exists. Figure 8 shows the estimated parameter matrix as distributions, where the value of k is divided by 10 for easier visualization.

Uncertainty on each parameter in the matrix is approximately equal, and all uncertainty bounds contain the true parameter value, indicated by a dashed line of the corresponding colour.

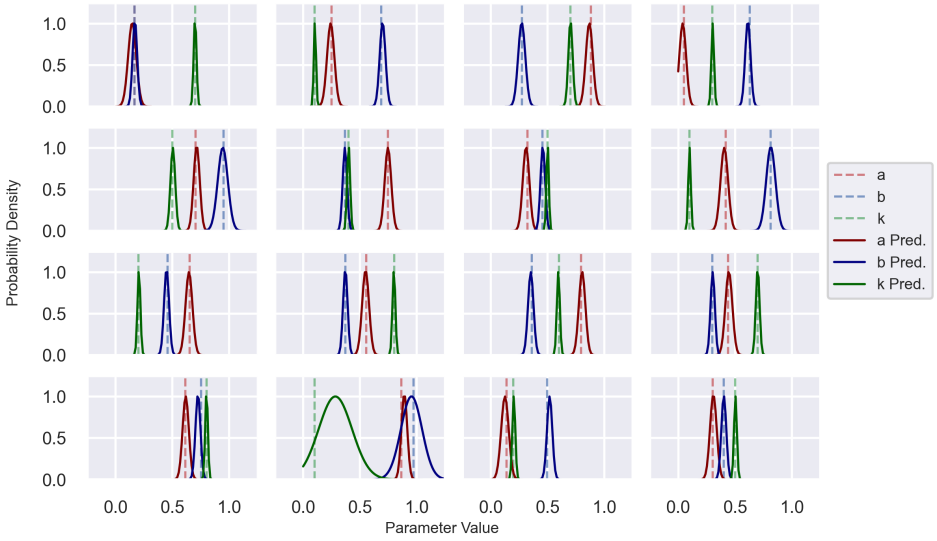


Figure 8: Plot of real and predicted ARX parameters for 4x4 system. Time delay k is divided by factor of 10 for visualization.

4.3. Model Validation

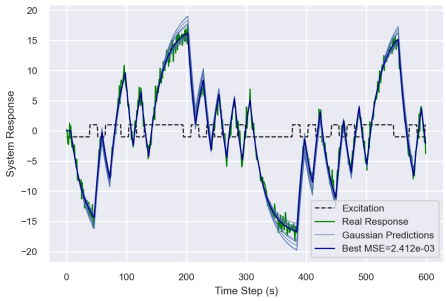
One of the most commonly used software packages for identification used by researchers is the MATLAB System Identification Toolbox [52]. Among the algorithms available in the toolbox, OE (output-error polynomial model) is the most relevant to the algorithm outlined in this paper. The parameters a and b are solved for using iterative nonlinear least-squares search-based updates which minimize the prediction error. The parameter uncertainty for a and b are estimated with covariance, whereas k is treated as a deterministic parameter. Equations 17 a, b and c show a real first order transfer function and the parameters plus uncertainty estimated using the MATLAB iterative method and the LSTM algorithm developed in this paper.

$$G(t)_{Real} = \frac{0.765485q^{-7}}{1 + 0.955585q^{-1}} + e(t) \tag{17a}$$

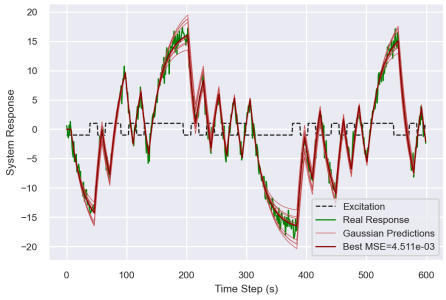
$$G(t)_{MATLAB} = \frac{(0.772 \pm 0.011)q^{-7}}{1 + (0.9553 \pm 0.0087)q^{-1}} + e(t) \tag{17b}$$

$$G(t)_{Python} = \frac{(0.772 \pm 0.017)q^{-(7.00 \pm 0.39)}}{1 + (0.955 \pm 0.023)q^{-1}} + e(t) \tag{17c}$$

The proposed model is validated against the MATLAB OE algorithm for coefficient of determination on parameter predictions, average mean squared error of the solution and solution time. 1000 samples of data, 300 s in length, are identified using each algorithm. Figure 9 shows a plot of prediction produced by the (a) MATLAB OE algorithm and (b) the proposed algorithm for the parameters shown in equations 17 a, b and c. Parameters are drawn from the Gaussian-distributed parameter range and plotted with the mean estimate. Mean squared error is calculated using the mean estimate and true system response.



(a) MATLAB SysID



(b) Proposed Algorithm

Figure 9: Comparison of MATLAB SysID and proposed algorithm at estimating a first order system with additive noise

Results shown in Figure 9 illustrates the trend observed across validation: MSE is consistently lower using MATLAB SysID but predictions from each algorithm look very similar when plotted. Gaussian-drawn predictions span a greater range using the proposed algorithm, underlining the result in equation 17 a and b that estimated uncertainty is higher using the proposed algorithm.

Table 1 compares the results of both algorithms on 1000 test samples

drawn from the training parameter range. The results agree strongly with expectations regarding the comparative speed of each algorithm. Prediction-error methods solve an optimization problem over all possible parameter pairs (a,b), whereas all optimization in the neural network occurs during training, and only a single pass of the network is needed to produce an estimate [53]. The slight difference in MSE is attributable to the fact that the MATLAB seeks to minimize the squared error between the real and predicted systems, while the proposed algorithm instead seeks to maximize the probability of estimated parameters [54].

Table 1: MATLAB SysID and Proposed Model Results for 1000 Samples

Model	Time (s)	R^2 (a)	R^2 (b)	R^2 (k)	MSE $(\times 10^{-4})$
MATLAB	95.28	0.9943	0.9984	0.9986	1.157
Proposed	3.544	0.9958	0.9994	0.9926	4.358

4.4. Model Generalization

It has been reported at great length that MPC performance is intractably connected to the treatment of disturbance in the internal model. This is attributable to the fact that the particular type and model of noise can introduce a mismatch between the controller prediction and the real system [55]. There exists no universal method for modeling uncertainty from process models - most uncertainty models are developed on a case-by-case basis [56]. In prediction-error cases, a true linear model is assumed to exist and noise is assumed to be white, the covariance matrix of the parameter matrix is assumed to describe parameter uncertainty introduced by these factors [57].

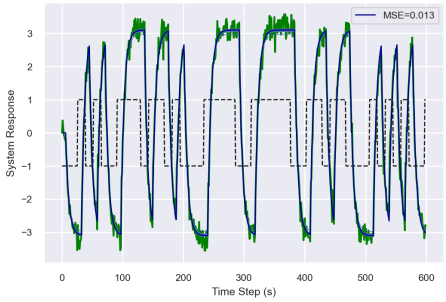
Understanding and properly demarcating model uncertainties based on their origin is an essential step in developing process models. While tolerance for aleatoric (white noise) uncertainty was handled explicitly in the training set, no consideration was made for epistemic uncertainty (lack of system knowledge) [58]. In the case of process modeling, one common source of epistemic uncertainty is choice of a first order model for a higher-order process. To validate the model developed in this paper on its inferential and epistemic estimating potential, we simulate a second-order process and compare the estimate and uncertainty bounds of the first-order estimates to the real model. Equations 18 a, b and c show the real second order model, the MATLAB OE solution and the proposed model solution.

$$G(t)_{Real} = \frac{0.799900q^{-6}}{1 + 0.270221q^{-1} + 0.479273q^{-2}} + e(t) \tag{18a}$$

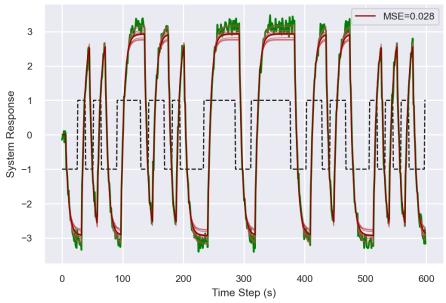
$$G(t)_{MATLAB} = \frac{(0.552 \pm 0.011)q^{-6}}{1 + (0.8222 \pm 0.0044)q^{-1}} + e(t) \tag{18b}$$

$$G(t)_{Python} = \frac{(0.545 \pm 0.013)q^{-(6.000 \pm 0.040)}}{1 + (0.814 \pm 0.013)q^{-1}} + e(t) \tag{18c}$$

Figure 10 shows the plot of the true and estimated transfer functions corresponding to equations 18 a, b and c.



(a) MATLAB SysID



(b) Proposed Algorithm

Figure 10: Comparison of MATLAB SysID and proposed algorithm at estimating a second order system with additive noise

MSE on predicted second-order systems shown in equation 18 is an order of magnitude higher than predicted first-order systems. Uncertainty on both predicted systems does not represent the fact that estimations seems to consistently under-estimate high-gain regions. This demonstrates a shortcoming of prediction-error methods which has partially carried over to the proposed algorithm. Uncertainty on estimated parameters represents the

sum of residuals on the optimized prediction, with the assumption that this error is stochastic and normally distributed.

4.5. Modeling With Disturbance

In the prediction error method, disturbance is treated exclusively as a single-component signal of high-frequency stochastic noise with zero mean. Prior work has shown that conventional identification algorithms are not reliable in instances where unmeasured disturbances significantly influence system output y(t) [59]. In these instances, sophisticated filters or regression models must be developed and integrated into the controller model [60]. Typically, this is implemented with a state and disturbance estimator, which explicitly handle disturbance estimation [61]. The rejection of disturbance results in a significant improvement in system identification efficiency. To investigate the possibility of integrating disturbance rejection into the controller model, a training set consisting of unmeasured system output w(t) is coupled with identifiable output x(t) to form output y(t). A sample set of w(t), x(t) and y(t) are shown in Figure 11.

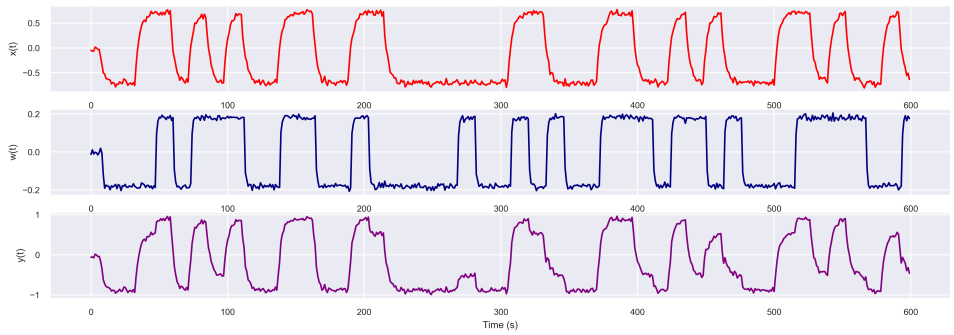


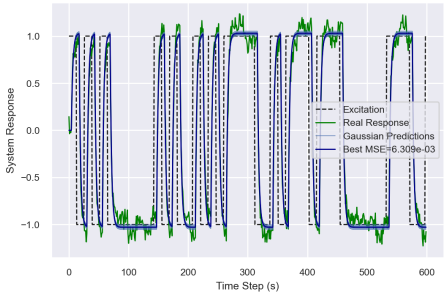
Figure 11: Plot of measured, 'unmeasured' and combined output from system

To attempt to incorporate disturbance rejection into the proposed network, a new model is trained on a set of 100,000 system responses with 5% additive 'unmeasured' disturbance and 5% Gaussian noise for 500 epochs. As was the case with the first model, the model from the epoch with the lowest validation error is retained. Table 2 compares the results of both MATLAB and the newly trained algorithm on 1000 test samples drawn from the training parameter range with unmeasured disturbance. MSE is the median of 1000 samples.

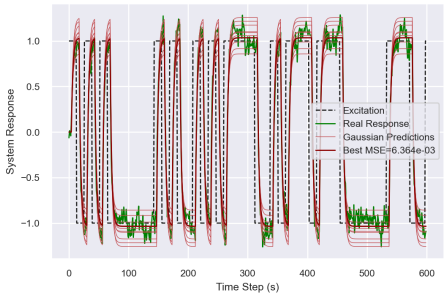
Table 2: MATLAB SysID and Proposed Model Results for 1000 samples of Unmeasured Disturbance

Model	R^2 (a)	R^2 (b)	R^2 (k)	MSE $(\times 10^{-2})$
MATLAB	0.9524	0.7927	0.5292	1.356
Proposed	0.9929	0.9260	0.9369	1.835

The coefficient of determination for parameters estimated with MATLAB OE algorithm is considerably lower than the proposed algorithm. This disparity does not translate to the mean squared error, evidently, as the mean squared error is higher for the proposed algorithm is 35% higher for the proposed algorithm. This, however is not indicative of successful system identification in the case of the MATLAB algorithm. Prediction-error methods rely on the assumption that the true system exists and is represented by the model, and that any stochasticity in the system response is normally-distributed. This is evidently not the case when structured disturbances are present in the data, and lead to deceptively low MSE despite having low prediction accuracy on the true system parameters. As shown above, the proposed algorithm may be trained with a-priori knowledge of the system behaviour, possibly accounting for structured disturbance and lead to better predictions with more accurate estimations of uncertainty. Figure 12 compares the estimating capacity of both identification algorithms on a system with 5% additive unmeasured disturbance.



(a) MATLAB SysID



(b) Proposed Algorithm

Figure 12: Comparison of MATLAB SysID and proposed algorithm at estimating a system with unmeasured disturbance

The low range of estimates made by the MATLAB system identification demonstrates that once the algorithm balances residuals on each side of the prediction, the predicted variance on estimated coefficient decreases significantly, giving the impression that noise is normally distributed, which is visibly not the case in this example. The wider distribution of parameter estimates in the proposed algorithm is consistent with results observed across the validation set. The proposed algorithm addresses this uncertainty by making more conservative estimates of model parameters and making predictions which approximate the true system.

5. Discussion - Second Generation Network

This section will discuss the results from the second round of research informed by the results of the first network. The changes made include reducing the number of parameters estimated by the network from a, b and k to just a and b. As mentioned in section 2.5, identification of the time constant k is typically addressed prior to implementing system identification algorithms. Time delay is typically solved for using the correlation-peak between input and output signals, which is a fast and widely-used technique [62]. Secondly, the network was adjusted to estimate parameters as a joint distribution. This was done to address over-training observed when a separate network was used to estimate each parameter and to better represent co-variant uncertainty between parameters a and b. Intuitively, if one coefficient in a model has extremely high uncertainty, it follows the second ought to account for this variation in its prediction. The length of input data and network structure remain the same, with the exception of the addition of two more linear auto-encoding units in the final layer of the network to account for the additional parameter.

5.1. System Training

100,000 samples were used to train and validate all models, with a training/validation split of 70/30. The same network architecture was utilized for identification of a, b and k: 100 units LSTM, 4 units with linear activation and 2 Lambda Distributions. All networks were used with default hyper-parameters where not specified. A mini-batch size of 256 was determined to lead to optimal performance and was used for all models. Models were trained using the Adam [49] optimizer with the default learning rate of 0.001. Figure 13 shows the training curves for the aforementioned network. Models were trained for 500 epochs. A minimum validation loss checkpoint was used, so that model weights leading to the lowest validation loss were stored and saved until the end of training. Training and validation loss curves follow closely, indicating that the validation and training sets are large and similar enough to avoid over-fitting.

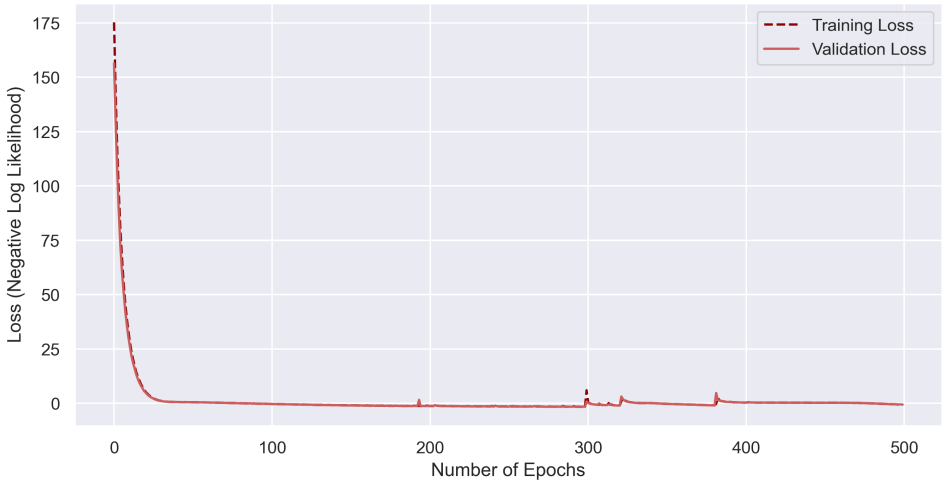


Figure 13: Training and validation loss for second network

5.2. Test Set Results

Figure 14 shows the results of the universal system model on unseen test sets of 1000 samples of unique input/output data. Testing parameters are all sampled from the same range as training and validation parameters. All validation coefficients of determination are above 0.995.

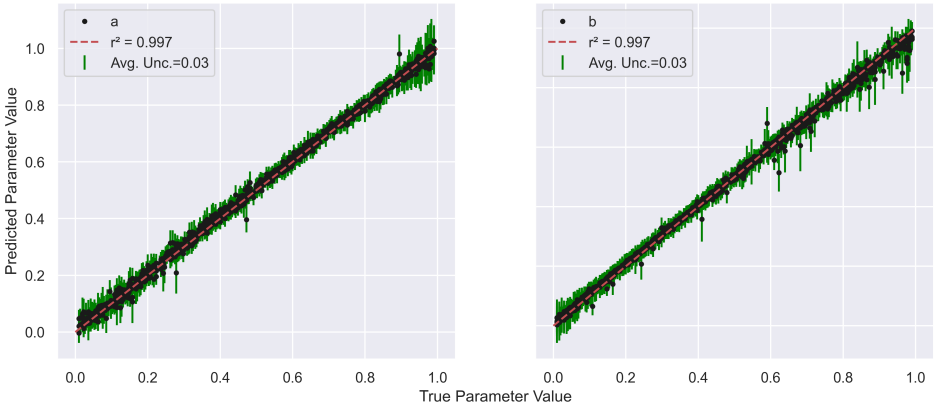


Figure 14: Plot of Gaussian predictions for parameters a and b with corresponding uncertainties

The primary goal of variational auto-encoding in neural networks regression is for uncertainty estimates to resemble the residuals / fitting errors of

the training set. Uncertainty bounds are relatively uniform across the middle of the parameter range, but increase dramatically at the periphery. This trend could be attributed to the fact that the training data becomes more sparse at the periphery. Augmenting the training set with parameters outside of this range (greater than 1 or less than 0) would be ideal, however, parameters outside of this range in most cases result in unstable linear systems [63]. A possible solution is to augment the training set with a higher number of training samples near the periphery of the parameter range to ensure an even distribution of uncertainty over the training space.

The average uncertainty in predictions from this system are shown on the plot to be 0.03, a few magnitudes greater than uncertainty observed in predictions in Figure 7. This suggests that over-training is partially abated by combining the parameters into a single distribution. To determine if estimated parameter uncertainty is linearly correlated with estimate residuals, a plot of uncertainty vs residual is created for 1000 new systems. A log-log plot is selected because a disproportional number of parameters are clustered around low magnitudes of error. Figure 15 and 16 shows the results from this analysis for parameter a and b, respectively. The color of each data point represents the magnitude of the true parameter value and the black line shows the linear regression line.

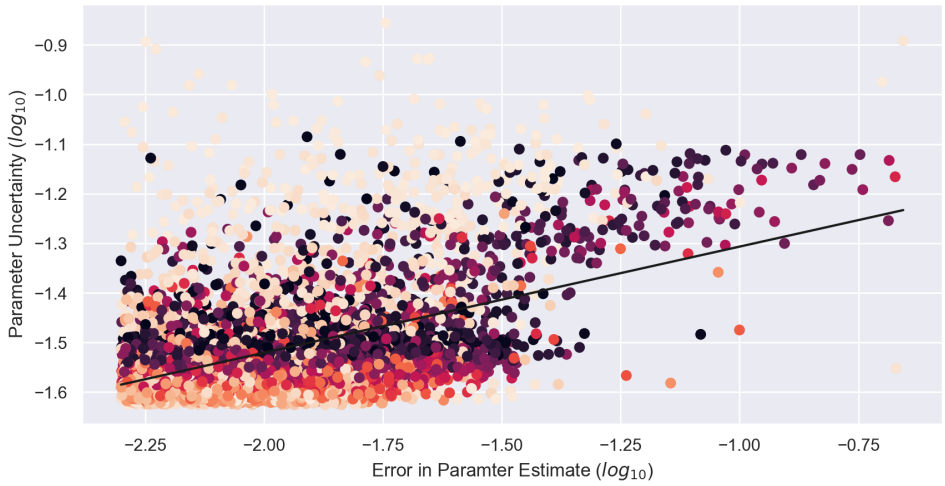


Figure 15: Predicted uncertainty vs prediction error for coefficient a. Data point color represents true parameter value in range (0,1). Black line is result of regression analysis.

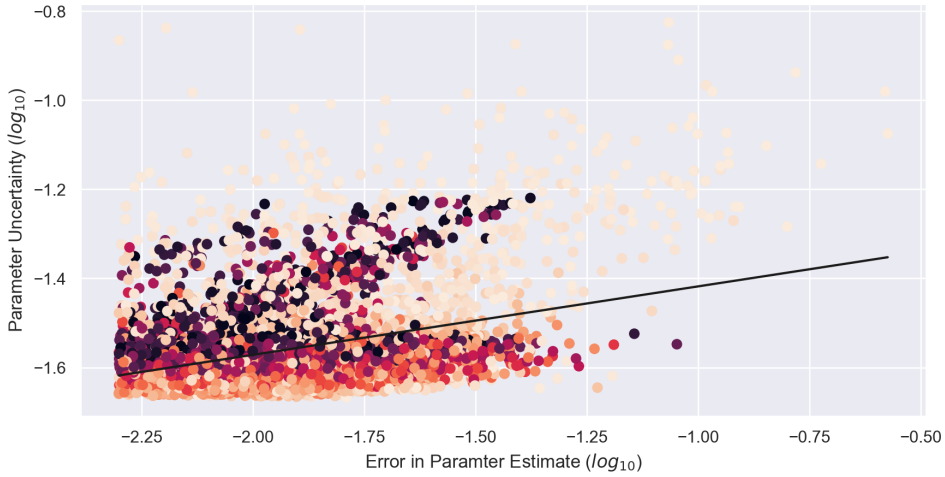
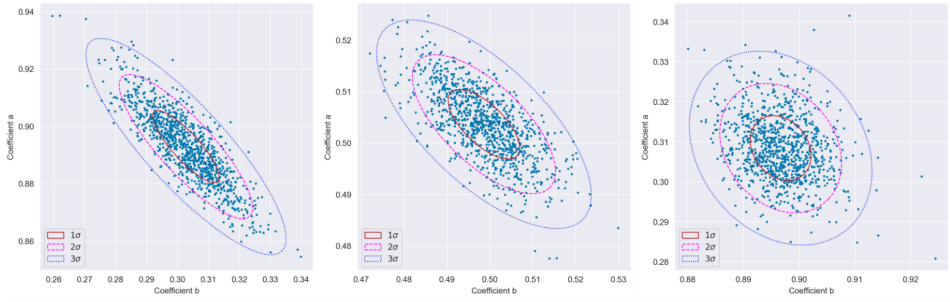


Figure 16: Predicted uncertainty vs prediction error for coefficient b. Data point color represents true parameter value in range (0,1). Black line is result of regression analysis.

From Figure 15, a positive relationship between error in parameter estimate and estimated uncertainty is observed, with the largest coefficients having the highest magnitude of error. Interestingly, a large population of estimates with low error but high uncertainty are observed on the left side of the plot. This indicates that while low-magnitude estimates of coefficient a are accurate, the predicted uncertainty is inexplicably high. From Figure 16, a positive relationship between error in parameter estimate and estimated uncertainty is again observed. Unlike the plot for coefficient a, the parameters with the highest prediction error and uncertainty are smaller in magnitude. This trend can be observed in Figure 14; predictions close to one for coefficient a have high disproportionately high estimates of uncertainty, whereas the inverse is observed for coefficient b. It is believed that this trend could be partially mitigated by increasing the number of training samples in these parameter ranges.

Confidence ellipses are a powerful tool which can be used visualize variance and covariance in a multi-dimensional distribution [64]. To visualize the coefficient distributions at different levels, three different systems are estimated from 1000 different system responses and plotted below in Figure 17. This figure is consistent with the trend observed during model validation: covariance between coefficients a and b are negative, and the magnitude of covariance decreases as coefficient a decreases and coefficient b decreases. The red, pink and blue lines inscribe the 1-σ, 2-σ and 3-σ levels of the joint distribution. The transfer function for the system shown in each figure is shown below the corresponding plot.



$$G(t) = \frac{0.3q^{-1}}{1 + 0.9q^{-1}} + e(t) \quad G(t) = \frac{0.5q^{-1}}{1 + 0.5q^{-1}} + e(t) \quad G(t) = \frac{0.9q^{-1}}{1 + 0.3q^{-1}} + e(t)$$

Figure 17: Prediction ellipses for three systems indicated below plot. Each blue dot represents an estimate from a unique data sequence

Uncertainty in the x and y direction closely matches that predicted by

the algorithm. This agrees with expectations, as each time a network pass is computed, a sample is randomly drawn from all possible parameter pairs in the joint distribution and returned with the uncertainty of the distribution. While this may make more sense for a stochastic process which has minor temporal fluctuations in system behaviour, it is not ideal in cases where an exact estimate of the system is required every time.

Table 3 shows the results of the new algorithm tested against the MATLAB System Identification package for 1000 new system responses. Solution speed is tested over five replicates. Solution speed is approximately three times faster than the prior algorithm. This agrees with expectations, as only a single neural network pass must be computed, instead of three separate passes for each model parameter. Coefficients of determination are slightly lower than the previous model. Error in parameter estimates is approximately four times greater than MATLAB. This small difference in internal model error did not lead to significantly different set-point tracking performance in closed-loop MPC experiments in MATLAB. In prior research pertaining to the acceptable range of parameter error in distillation columns by Tufa and Chong (2016), 70% overall integral error (OIE) was determined to an acceptable threshold in model plant mismatch for set-point tracking of a step change [65]. OIE in all closed-loop simulations remained significantly below 1%.

Table 3: MATLAB SysID and Proposed Model Results for 1000 samples of Unmeasured Disturbance. Time averaged over five replicates.

Model	Time [s]	R^2 (a)	Error (a)	R^2 (b)	Error (b)
MATLAB	89.4 ($\pm$ 0.9)	0.9997	0.0035	0.9997	0.0031
Proposed	1.16 ($\pm$ 0.04)	0.9980	0.0140	0.9971	0.0140

5.3. Model Generalization

Multiple techniques for closed-loop identification are employed in industry. For systems under MPC control, there are generally three main methods: the direct approach, where the input u and output y are treated as an open-loop system, and changes to the reference signal r are ignored; the indirect approach, where changes in the reference signal r are used as the input and y as the output; the joint input-output approach where the output y and and input u are considered to be outputs of a system driven by changes in the reference signal r [66]. If unsatisfactory system disturbance is achieved, a dither signal can be added to the system through input u

or to output y. The reference signal r is the feedback of MPC control. In prior work, simulations of random systems have demonstrated that a random process with memory is a good surrogate for the response signal of an MPC with zero-mean [67]. Simulating a true closed-loop system and implementing the algorithm online is beyond the scope of this work, so random process signals as described in Section 3.1 are used to approximate closed-loop response of an MPC controller. 1000 system responses are generated using the random process signals and the network trained on PRBS is used to predict the system parameters; Figure 18 shows the results.

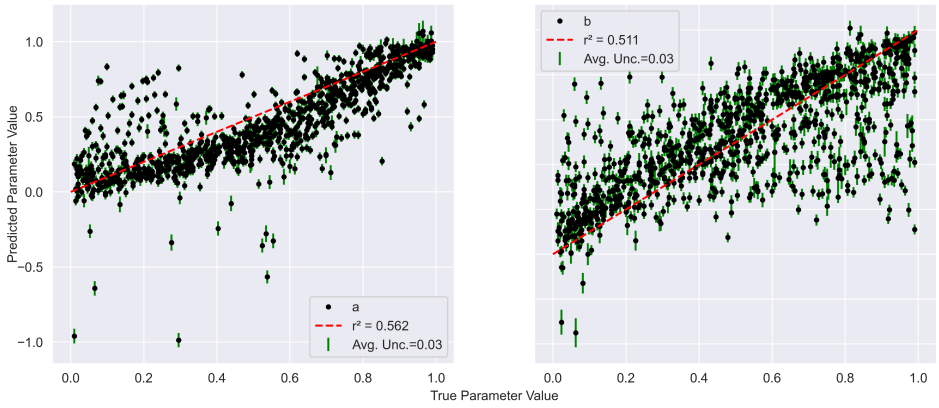


Figure 18: Validating a network trained on PRBS on 1000 samples of random process system responses to test generalization ability of network

Unsurprisingly, the network generalizes poorly to the new type of data. To evaluate the complexity of the random process compared to the PRBS signal, spectral entropy is used to estimate signal complexity: a method commonly used in signal processing [68]. With entropy evaluated with base e, the average entropy of the PRBS signal used is 4.8 while the random process is 5.5, averaged over 10 replicates. This supports the qualitative observation that the random process is more rich and extracts more dynamic behaviour from the system. Interestingly, empirical changes to a training set, like increasing the informativeness, is not commonly utilized in the literature. This is likely because machine learning problems are typically formulated around an existing dataset with prescribed features. Changes to most machine learning datasets are thus typically relegated to linear transformation of existing features [69]. To test if the drop in network accuracy is a result of increasing entropy in the perturbation signal, a new network is trained for 500 epochs on 100,000 samples of random process

system perturbations. Figure 19 shows the performance of the network on a validation set of 1000 samples of random process data.

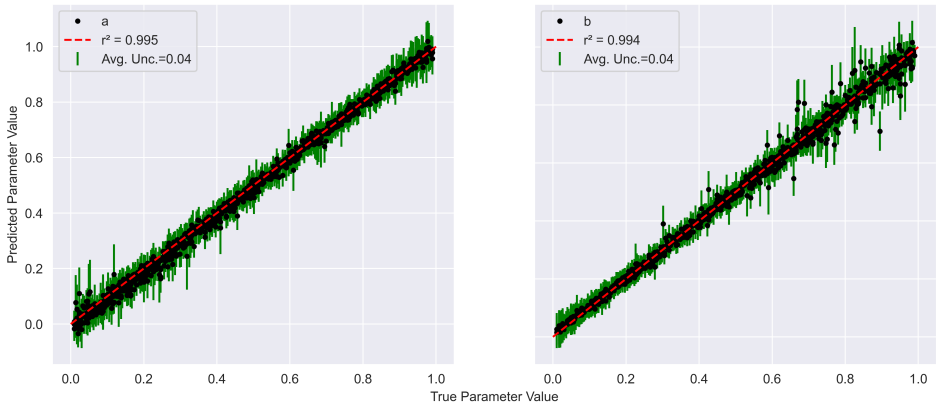


Figure 19: Validating a network trained on random process perturbations on 1000 samples of random process system responses

Predicted uncertainty is slightly higher and coefficients of determination are slightly lower for this network, as opposed to the network trained on PRBS data. This agrees with the expectation that more epochs are necessary to achieve the same predictive capacity as the PRBS network, due to the increased entropy of the new training data. Next, to test the generalization capacity of this network, 1000 system responses are generated using PRBS perturbations and the random-process-trained network is used as a predictor. Figure 20 shows the results of the network predicting on 1000 new random process system responses.

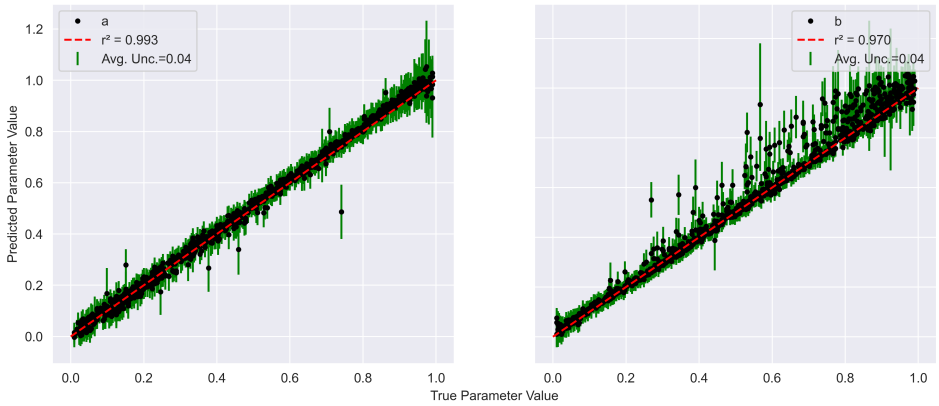


Figure 20: Validating a network trained on random process on 1000 samples of PRBS responses to test generalization ability of network

The coefficients of determination for this validation set are 0.993 and 0.970 for coefficients a and b, respectively, with similar uncertainty bounds to the first validation set. These results strongly support the hypothesis that increasing the entropy of the training set increases the ability of the network to generalize to less complex perturbation signals. This has significant implications for the closed-loop usefulness of the network. There is far less control over the frequency and and shape of system-identification in closed-loop compared to open-loop techniques. If an algorithm can infer to less complex signals during validation, a training set only needs to be designed for the complex end of the input signal spectrum.

5.4. Limitations of Variational Auto-encoders

Variational auto-encoders have seen increased application as inferential predictors of uncertainty. According to the definition outlined in Equation 11, this uncertainty is function of the back-propagated divergence between the predicted and true label on the right-hand-side of the network. The resultant trend between increasing the dimension of training data and the accuracy of uncertainty bounds is thus at first glance counter-intuitive. Prior research has remarked that in some extraneous cases, uncertainty is underestimated as a result of a sparse training set [70]. Use of the Kullback-Leibler divergence also assumes that the latent representation of the input data (the output from the LSTM) is drawn from the same distribution as the training set. For this reason, variational auto-encoders are commonly referred to as adversarial inference tools, instead of uncertainty predictors,

as that definition is misleading when uncertainty is thought of as a lack of knowledge [71]. This phenomenon can be observed in Figure 18, where uncertainty remains low despite the fact the network is predicting on input data outside of thee training set. This makes the variational auto-encoder a poor predictor of unconventional system behaviour, and cannot be used independently to reject a predicted system structure.

6. Recommendations and Future Work

The advantages and disadvantages of the proposed system identification algorithm has been discussed at length. It is recommended that future work that the suggestions regarding the size and diversity of the training set are heeded and that a more thorough training set is devised to give more accurate uncertainty bounds on predictions. The most significant evidence of a successful system identification algorithm which was not brought forth in this work was validation on a set of real data. It is recommended in the future that the network is tested on real data and implemented in real-time closed-loop control to determine if it offers any practical advantages in practice. Another avenue which was not investigated in this work but holds promise is using the network to provide initial estimates to an ordinary prediction-error algorithm, this would likely significantly decrease convergence time and address problems with initial conditions observed with this algorithm.

7. Conclusion

This work has shown that a recurrent neural network structure with variational auto-encoders is capable of identifying multidimensional linear systems with meaningful uncertainty bounds. Deep neural networks are faster to implement than least squares algorithms, and could be combined with least squares to increase prediction accuracy while maintaining prediction speed. It was demonstrated that the algorithm developed in this paper is ninety times faster than the MATLAB OE function. Aleatoric and epistemic uncertainty were investigated independently and it was shown that predictions and uncertainty bounds are interpolative, meaning that a training set must necessarily include anomalous data in order to produce representative uncertainty bounds. With deep neural network models trained on representative datasets, it is possible to develop fast algorithms which can predict linear system parameters from sets of input/output data. More specifically, this algorithm can be used in an MPC structure, updating internal system model to respond to temporal changes in dynamics. In future investigations, we hope to improve the accuracy of the algorithm, implement on higher dimension MIMO systems and use to identify real systems from experimental data.

References

[1] I. Yassin, M. Taib, R. Adnan, Recent advancements & methodologies in system identification: A review, 2013.

[2] C. E. García, D. M. Prett, M. Morari, Model predictive control: Theory and practice—a survey, Automatica 25 (1989) 335 – 348. URL: http://www.sciencedirect.com/science/article/pii/0005109889900022. doi:https://doi.org/10.1016/0005-1098(89)90002-2.

[3] B. Mehta, Y. Reddy, Industrial process automation systems: Design and implementation, 2014.

[4] S. V. Raković, Model predictive control: Classical, robust, and stochastic [bookshelf], IEEE Control Systems Magazine 36 (2016) 102–105. doi:10.1109/MCS.2016.2602738.

[5] Y. Zhu, System identification for process control: Recent experience and outlook, IFAC Proceedings Volumes 39 (2006) 20 – 32. URL: http://www.sciencedirect.com/science/article/pii/S1474667015352393. doi:https://doi.org/10.3182/20060329-3-AU-2901.00003, 14th IFAC Symposium on Identification and System Parameter Estimation.

[6] W. MacArthur, C. Zhan, A practical global multi-stage method for fully automated closed-loop identification of industrial processes, Journal of Process Control 17 (2007) 770–786. doi:10.1016/j.jprocont.2007.04.003.

[7] Y. Zhu, Tai-ji id automatic closed-loop identification package for model based process control, IFAC Proceedings Volumes 33 (2000) 509 – 513. URL: http://www.sciencedirect.com/science/article/pii/S1474667017398014. doi:https://doi.org/10.1016/S1474-6670(17)39801-4, 12th IFAC Symposium on System Identification (SYSID 2000), Santa Barbara, CA, USA, 21-23 June 2000.

[8] X. Liu, Y. Zhu, The asymptotic method for the identification of errors-in-variables systems, in: 2016 35th Chinese Control Conference (CCC), 2016, pp. 1952–1957. doi:10.1109/ChiCC.2016.7553654.

[9] Tai-ji online ProcessDoctor–Tai-Ji, http://www.taijicontrol.com/onlineintro.htm, ???? Accessed: 2020-11-15.

[10] J. O. Ramsay, G. Hooker, D. Campbell, J. Cao, Parameter estimation for differential equations: a generalized smoothing approach, Journal of the Royal Statistical Society: Series B (Statistical Methodology) 69 (2007) 741–796. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9868.2007.00610.x. doi:https://doi.org/10.1111/j.1467-9868.2007.00610.x. arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-9868.2007.00610.

[11] T. A. Tutunji, Parametric system identification using neural networks, Applied Soft Computing 47 (2016) 251 – 261. URL: http://www.sciencedirect.com/science/article/pii/S1568494616302137. doi:https://doi.org/10.1016/j.asoc.2016.05.012.

[12] I. Gabrijel, A. Dobnikar, On-line identification and reconstruction of finite automata with generalized recurrent neural networks, Neural Networks 16 (2003) 101 – 120. URL: http://www.sciencedirect.com/science/article/pii/S0893608002002216. doi:https://doi.org/10.1016/S0893-6080(02)00221-6.

[13] M. Fei, J. Zhang, H. Hu, T. Yang, A novel linear recurrent neural network for multivariable system identification, Transactions of the Institute of Measurement and Control 28 (2006) 229–242. URL: https://doi.org/10.1191/0142331206tim171oa. doi:10.1191/0142331206tim171oa. arXiv:https://doi.org/10.1191/0142331206tim171oa.

[14] S. Genc, Parametric system identification using deep convolutional neural networks, in: 2017 International Joint Conference on Neural Networks (IJCNN), 2017, pp. 2112–2119. doi:10.1109/IJCNN.2017.7966110.

[15] N. C. Thompson, K. Greenewald, K. Lee, G. F. Manso, The computational limits of deep learning, 2020. arXiv:2007.05558.

[16] R. A. Abou Jeyab, Y. P. Gupta, J. R. Gervais, P. A. Branchi, S. S. Woo, Constrained multivariable control of a distillation column using a simplified model predictive control algorithm, Journal of process control 11 (2001) 509–517.

[17] J. E. Castaño, J. A. Patiño, J. J. Espinosa, Model identification for control of a distillation column, in: IX Latin American Robotics Symposium and IEEE Colombian Conference on Automatic Control, 2011 IEEE, 2011, pp. 1–5. doi:10.1109/LARC.2011.6086832.

[18] Y. Zhu, Multivariable process identification for mpc: the asymptotic method and its applications, Journal of Process Control 8 (1998) 101–115. URL: https://www.sciencedirect.com/science/article/pii/S0959152497000358. doi:https://doi.org/10.1016/S0959-1524(97)00035-8.

[19] Y. Zhu, E. A. A. Books, Multivariable System Identification for Process Control, Elsevier Science [Imprint], San Diego, 2001.

[20] Y. Zhu, M. Van Wijek, E. Janssen, T. Graaf, K. Van Aalst, L. Kieviet, Crude unit identification for mpc using asym method, volume 5, IEEE, 1997, pp. 3395–3399 vol.5.

[21] B. S. Dayal, J. F. MacGregor, Identification of finite impulse response models: methods and robustness issues, Industrial & engineering chemistry research 35 (1996) 4078–4090.

[22] Y. C. Zhu, A. C. P. M. Backx, P. Eykhoff, Multivariable process identification based on frequency domain measures, IEEE, 1991, pp. 303–308 vol.1.

[23] L. Ljung, Prediction error estimation methods, Circuits, Systems, and Signal Processing 21 (2002) 11–21. doi:10.1007/BF01211648.

[24] H. Tonnang, A. Olatunbosun, I. NNC, Neural network controller for a crude oil distillation column, ARPN J. Eng. Appl. Sci. 5 (2010).

[25] D. Ahmed, A. Khalaf, Artificial neural networks controller for crude oil distillation column of baiji refinery, Journal of Chemical Engineering & Process Technology 07 (2015). doi:10.4172/2157-7048.1000272.

[26] R. Diversi, R. Guidorzi, U. Soverini, Identification of arx and ararx models in the presence of input and output noises, European Journal of Control - EUR J CONTROL 16 (2010) 242–255. doi:10.3166/ejc.16.242-255.

[27] J. Kon, Y. Yamashita, T. Tanaka, A. Tashiro, M. Daiguji, Practical application of model identification based on arx models with transfer functions, Control Engineering Practice 21 (2013) 195 – 203. URL: http://www.sciencedirect.com/science/article/pii/S0967066112002134. doi:https://doi.org/10.1016/j.conengprac.2012.09.021.

[28] M. H. Hassoun, et al., Fundamentals of artificial neural networks, MIT press, 1995.

[29] L. R. Medsker, L. Jain, Recurrent neural networks, Design and Applications 5 (2001).

[30] R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks, in: International conference on machine learning, 2013, pp. 1310–1318.

[31] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735–80. doi:10.1162/neco.1997.9.8.1735.

[32] W. J. Maddox, P. Izmailov, T. Garipov, D. P. Vetrov, A. G. Wilson, A simple baseline for bayesian uncertainty in deep learning, Advances in Neural Information Processing Systems 32 (2019) 13153–13164.

[33] A. Urbina, S. Mahadevan, T. L. Paez, Quantification of margins and uncertainties of complex systems in the presence of aleatoric and epistemic uncertainty, Reliability Engineering & System Safety 96 (2011) 1114–1125.

[34] D. P. Kingma, M. Welling, Auto-encoding variational bayes, arXiv preprint arXiv:1312.6114 (2013).

[35] M. Rosca, B. Lakshminarayanan, D. Warde-Farley, S. Mohamed, Variational approaches for auto-encoding generative adversarial networks, arXiv preprint arXiv:1706.04987 (2017).

[36] J. R. Hershey, P. A. Olsen, Approximating the kullback leibler divergence between gaussian mixture models, in: 2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07, volume 4, IEEE, 2007, pp. IV–317.

[37] T. Van Erven, P. Harremos, Rényi divergence and kullback-leibler divergence, IEEE Transactions on Information Theory 60 (2014) 3797–3820.

[38] X. Yang, Understanding the variational lower bound, 2017.

[39] T. Söderström, P. Stoica, System identification, 1988.

[40] D. E. Rivera, K. S. Jun, An integrated identification and control design methodology for multivariable process system applications, IEEE Control Systems Magazine 20 (2000) 25–37. doi:10.1109/37.845036.

[41] J. S. Conner, D. E. Seborg, An evaluation of mimo input designs for process identification, Industrial & Engineering Chemistry Research 43 (2004) 3847–3854. URL: https://doi.org/10.1021/ie034068+. doi:10.1021/ie034068+. arXiv:https://doi.org/10.1021/ie034068+.

[42] A. M. Doust, F. Shahraki, J. Sadeghi, Simulation, control and sensitivity analysis of crude oil distillation unit, 2012.

[43] J. Chung, K. Kastner, L. Dinh, K. Goel, A. Courville, Y. Bengio, A recurrent latent variable model for sequential data, 2016. arXiv:1506.02216.

[44] J. Yu, M. W. Lam, S. Hu, X. Wu, X. Li, Y. Cao, X. Liu, H. Meng, Comparative Study of Parametric and Representation Uncertainty Modeling for Recurrent Neural Network Language Models, in: Proc. Interspeech 2019, 2019, pp. 3510–3514. URL: http://dx.doi.org/10.21437/Interspeech.2019-1927. doi:10.21437/Interspeech.2019-1927.

[45] D. Park, Y. Hoshi, C. C. Kemp, A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder, IEEE Robotics and Automation Letters 3 (2018) 1544–1551. doi:10.1109/LRA.2018.2801475.

[46] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/, software available from tensorflow.org.

[47] D. P. Kingma, M. Welling, An introduction to variational autoencoders, Foundations and Trends® in Machine Learning 12 (2019) 307–392. URL: http://dx.doi.org/10.1561/2200000056. doi:10.1561/2200000056.

[48] C. Sønderby, T. Raiko, L. Maaløe, S. Sønderby, O. Winther, How to train deep variational autoencoders and probabilistic ladder networks (2016).

[49] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2017. arXiv:1412.6980.

[50] J. Platt, et al., Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods, Advances in large margin classifiers 10 (1999) 61–74.

[51] C. Cremer, X. Li, D. Duvenaud, Inference suboptimality in variational autoencoders, 2018. arXiv:1801.03558.

[52] L. Ljung, Matlab system identification toolbox, R2020b. URL: *https : //www.mathworks.com/help/pdf*$_d$*oc/ident/ident*$_u$*g.pdf*, the MathWorks, Natick, MA, USA.

[53] K. Astrom, Maximum likelihood and prediction error methods, IFAC Proceedings Volumes 12 (1979) 551–574. URL: https://www.sciencedirect.com/science/article/pii/S1474667017539762. doi:https://doi.org/10.1016/S1474-6670(17)53976-2, tutorials presented at the 5th IFAC Symposium on Identification and System Parameter Estimation, Darmstadt, Germany, September.

[54] E. Nachmani, Y. Bachar, E. Marciano, D. Burshtein, Y. Be'ery, Near maximum likelihood decoding with deep learning, arXiv preprint arXiv:1801.02726 (2018).

[55] J. K. Huusom, N. K. Poulsen, S. B. Jørgensen, J. B. Jørgensen, Tuning siso offset-free model predictive control based on arx models, Journal of Process Control 22 (2012) 1997 – 2007. URL: http://www.sciencedirect.com/science/article/pii/S0959152412001941. doi:https://doi.org/10.1016/j.jprocont.2012.08.007.

[56] L.-K. Chen, A. G. Ulsoy, Identification of a driver steering model, and model uncertainty, from driving simulator data, Journal of dynamic systems, measurement, and control 123 (2001) 623–629.

[57] L. Ljung, Matlab system identification toolbox reference, R2020b. URL: *https : //www.mathworks.com/help/pdf*$_d$*oc/ident/ident*$_r$*ef.pdf*, the MathWorks, Natick, MA, USA.

[58] A. Fanfarillo, Quantifying uncertainty in source term estimation with tensorflow probability, in: 2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC), 2019, pp. 1–6. doi:10.1109/UrgentHPC49580.2019.00006.

[59] D. Kim, System identification for building thermal systems under the presence of unmeasured disturbances in closed loop operation: Lumped disturbance modeling approach, Building and Environment 107 (2016) 169–180. doi:10.1016/j.buildenv.2016.07.007.

[60] R. Isermann, M. Münchhof, Identification of Dynamic Systems: An Introduction with Applications, 2011. doi:10.1007/978-3-540-78879-9.

[61] M. Mohammadkhani, F. Bayat, A. A. Jalali, Design of explicit model predictive control for constrained linear systems with disturbances, International Journal of Control, Automation and Systems 12 (2014) 294–301. doi:10.1007/s12555-013-0058-0.

[62] Lei Zhang, Xiaolin Wu, On cross correlation based-discrete time delay estimation, in: Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005., volume 4, 2005, pp. iv/981–iv/984 Vol. 4. doi:10.1109/ICASSP.2005.1416175.

[63] M. Galrinho, N. Everitt, H. Hjalmarsson, Arx modeling of unstable linear systems, Automatica (Oxford) 75 (2017) 167–171.

[64] M. Friendly, G. Monette, J. Fox, Elliptical insights: Understanding statistical methods through elliptical geometry, Statistical Science 28 (2013). doi:10.1214/12-STS402.

[65] L. D. Tufa, Z. K. Chong, Effect of model plant mismatch on mpc performance and mismatch threshold determination, Procedia Engineering 148 (2016) 1008–1014. doi:10.1016/j.proeng.2016.06.518.

[66] E. de Klerk, I. Craig, Closed-loop system identification of a mimo plant controlled by a mpc controller, IEEE AFRICON. 6th Africon Conference in Africa, 1 (2002) 85–90 vol.1.

[67] Y. Wang, S. Boyd, Fast model predictive control using online optimization, Control Systems Technology, IEEE Transactions on 18 (2010) 267 – 278. doi:10.1109/TCST.2009.2017934.

[68] P. Flegner, K. Ján, Evaluation of sensor signal processing methods in terms of information theory, Acta Polytechnica 58 (2018) 339–345. doi:10.14311/AP.2018.58.0339.

[69] R. Novak, Y. Bahri, D. Abolafia, J. Pennington, J. Sohl-Dickstein, Sensitivity and generalization in neural networks: an empirical study (2018).

[70] K. Gundersen, A. Oleynik, N. Blaser, G. Alendal, Semi conditional variational auto-encoder for flow reconstruction and uncertainty quantification from limited observations (2020).

[71] Y. Yang, P. Perdikaris, Adversarial uncertainty quantification in physics-informed neural networks, Journal of Computational Physics 394 (2019). doi:10.1016/j.jcp.2019.05.027.

Appendix B

Code for Generating Results

Two main classes are included:

Signal: generates system responses and linear combinations for system training

Model: generates models and predictions

Additional code for generating results can be found at :

https://github.com/rballachay/RNNSysID

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 30 16:25:30 2020

@author: RileyBallachay
"""
import os
import numpy as np
import random
import math
import scipy
from pylfsr import LFSR
import matplotlib.pyplot as plt
from pathlib import Path
import scipy.signal as signal
from mpl_toolkits.mplot3d import Axes3D
from scipy.signal import max_len_seq
from joblib import Parallel, delayed
import control as control
import time

class Signal:
    """
    Class that produces input signals and output of system response.

    Uses either a wave-based input signal created using hann signal
    or a pseudo-binary input signal with 2-10 steps in the signal window.
    System response can currently only be simulated as first order plus time
    delay SISO. In the future, MIMO will also be utilized.

    The purpose of this class is to produce random input and output signals
    from simulated systems that can be used to train the class Model, which
    will then predict the system parameters of other first-order systems.

    Parameters
    ----------
    numTrials : int, default=100
        Integer ideally bounded between 10-1000. Warning, simulation can take
        a very long time if greater than 1000. Will determine the number
        of simulations produced.

    nstep : int, default=100
        Number of steps in time frame. Majority of indexing used in signal
        and response depend on the index of the time and input signal array,
        so this can be more important than timelength.

    timelength : float, default=100.
        The length of the interpreted input/output signal data. In order to
        scale time constants appropriately, must be in seconds. Need more robust
```

```
    scaling method for absolute value of system parameters.

trainFrac : float, default=0.7
    The fraction of data used for validation/testing after fitting model.
    If model is only used to predict, trainFrac is forced to 1.

stdev : float, default=5.
    The standard deviation of Gaussian noise applied to output signal data
    to simulate error in real-world measurements.

Attributes
----------
random_signal
    Generates hann windows with varying width and amplitude and appends to
    produce a pseudo-random wave sequence.

PRBS
    Generates a pseudo-random binary signal with varying width. Frequency is
    random, depends on probability switch. 10% probability that the signal
    changes sign every time step. Average step width of 6.5.

plot_parameter_space
    Produces 3D plot of all simulated parameters (a,b,k)

gauss_noise
    Adds gaussian noise to input sequence and returns array with noise.

find_nearest
    odeint is built to take constant or functions as attributes. In this cas
    u is an array, so find_nearest is used to find the nearest value in u ar

FOmodel
    First order plus time delay model in state space format.

training_simulation
    Iterates over the input parameter space and produces simulations which
    will subsequently be used to train a model.

preprocess
    Separates data into training and validation sets and reshapes for input
    to the GRU model.

simulate_and_preprocess
    Function which produces data to be used directly in prediction. Cannot b
    used if data is to be used in training.

"""

def __init__(self,inDim,outDim,numTrials,trainFrac=0.7,numPlots=5,stdev=5):
    self.numTrials = numTrials
```

```
        self.nstep = 3*50*(inDim+outDim)
        self.timelength = self.nstep
        self.trainFrac = trainFrac
        self.valFrac = 1-trainFrac
        self.numPlots = numPlots
        self.stdev=stdev
        self.special_value=-99
        self.startTime=time.time()
        self.inDim = inDim
        self.outDim = outDim
        self.maxLen = 5
        self.length = 300
        self.STDEVS = []
        self.pd = str(Path(os.getcwd()).parent)
        if self.inDim>1:
            self.trim = 300
        else:
            self.trim=0

    def PRBS_parameterization(self,numRegisters=9,force_maximum=True):
        """This function serves to determine the best PRBS
        parameters to allow for identification. Based on
        work from Rivera and Jung, 2000: "An integrated
        identification and control design methodology
        for multivariable process system applications"

        T_sw: Switching time
        nr: Number of registers
        tau_L: Minimum time constant
        tau_H: Maximum time constant
        alpha: Closed-loop response speed
        """
        try:
            if not(force_maximum):
                tau_L = self.a_possible_values[0]
                tau_H = self.a_possible_values[1]
            else:
                tau_L = 0.01 ; tau_H = 0.99
        except:
            print("You haven't initialized a system, continuting with default PR
            tau_L = 0.01 ; tau_H = 0.99

        # Five times the settling time is 99% settled
        T_sw = int(278*tau_H/20)

        """Yields MLSequence of 511. Sequence repeats after
        nr*T_sw sampling intervals.
        """
        nr = numRegisters
        Ns = 2**nr-1
```

```
        return T_sw,Ns*T_sw


    def PRBS(self):
        """Returns a pseudo-random binary sequence
        which ranges between -1 and +1. This algorithm
        assumes the maximum time constant is 10, and uses
        the time constant to determine the """

        sample,max_len = self.PRBS_parameterization()

        if self.length>max_len:
            self.length=max_len

        L = LFSR(fpoly=[9,5],initstate ='random',verbose=False)
        L.runKCycle(int(np.floor(self.length/sample))+1)
        seq = L.seq
        PRBS = np.zeros(self.length)
        for i in range(0,int(self.length/sample)+1):
            if seq[i]==0:
                seq[i]=-1

            if sample*i+sample>=self.length:
                PRBS[sample*i:] = seq[i]
                break

            PRBS[sample*i:sample*i+sample]=seq[i]

        return PRBS

    def random_process(self):

        sample,max_len = self.PRBS_parameterization()

        if self.length>max_len:
            self.length=max_len

        # random signal generation

        a_range = [-1,1]
        a = np.random.rand(self.length) * (a_range[1]-a_range[0]) + a_range[0] #

        b_range = [10, 15]
        b = np.random.rand(self.length) *(b_range[1]-b_range[0]) + b_range[0] #
        b = np.round(b)
        b = b.astype(int)

        b[0] = 0
```

```
    for i in range(1,np.size(b)):
        b[i] = b[i-1]+b[i]

    # Random Signal
    i=0
    random_signal = np.zeros(self.length)
    while b[i]<np.size(random_signal):
        k = b[i]
        random_signal[k:] = a[i]
        i=i+1
    return random_signal


def plot_parameter_space(self,x,y,z,trainID,valID):
    """This function plots the parameter space for a first
    order plus time delay model in 3D coordinates"""
    x=np.array(x); y=np.array(y); z=np.array(z)
    figgy = plt.figure(dpi=200)
    ax = Axes3D(figgy)
    xT = x[trainID]; xV = x[valID]
    yT = y[trainID]; yV = y[valID]
    zT = z[trainID]; zV = z[valID]
    ax.scatter(xT,yT,zT,c='g',label="Training Data")
    ax.scatter(xV,yV,zV,c='purple',label="Validation Data")
    ax.set_xlabel("A (Denominator)")
    ax.set_ylabel("B (Numerator)")
    ax.set_zlabel("K (Time-Shift Operator)")
    ax.legend()

def gauss_noise(self,array,stdev):
    """Generate gaussian noise with mean and standard deviation
    of 5% of the maximum returned value."""
    # If the array has 2 dimensions, this will capture it
    # Otherwise, it will evaluate the length of 1D array
    if stdev=='variable':
        stdev = abs(np.random.normal(5,1))
        self.STDEVS.append(stdev)
    noise = np.random.normal(0,(stdev/100)*np.max(abs(array)),array.shape)
    return array+noise

def serialized_checkpoint(self,iteration):
    """Checkpoint which is called when
    .2 fraction of the way thru the data"""
    try:
        checkTime = float(time.time()) - self.Lasttime
    except:
        checkTime = float(time.time()) - self.startTime

    self.Lasttime = float(time.time())
    checkpoint = int(100*iteration/self.numTrials)
```

```
        print("Produced %i%% of the serialized data" %checkpoint)
        print("Estimated Time Remaining: %.1f s\n" % (checkTime*(100-checkpoint)

    def add_disturbance(self):

        a,_,_ = self.uArray.shape
        self.randint = np.random.randint(a)
        # Create first PRBS outside of loop
        u = self.uArray[self.randint,:,:]
        # The transfer function from the 2nd input to the 1st output is
        # (3z + 4) / (6z^2 + 5z + 4).
        # num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
        # Iterate over each of the output dimensions and
        # add to numerator
        allY=np.zeros((self.nstep,self.outDim))
        for j in range(0,self.outDim):
            # Iterate over each of the input dimensions
            # and add to the numerator array
            numTemp = [random.choice(self.b_real_params) for i in range(self.inD
            denTemp = [[1.,-random.choice(self.a_real_params)[0]] for i in range
            kVals = [random.choice(self.k_real_params)[0] for i in range(self.in

            bigU=np.transpose(u)
            uSim=np.zeros_like(bigU)
            for (idx,row) in enumerate(bigU):
                try:
                    uSim[idx,:] = np.concatenate((np.zeros(kVals[idx]),row[:-kVa
                except:
                    uSim[idx,:] = row
            numTemp=np.array([numTemp]);denTemp=np.array([denTemp])
            sys = control.tf(numTemp,denTemp,1)
            _,y,_ = control.forced_response(sys,U=uSim,T=self.t)

            steps = int(self.nstep/self.length)
            l=self.length
            for step in range(0,steps):
                allY[l*step:l*(step+1),j] = self.gauss_noise(y[l*step:l*(step+1)

        return allY


    def y_map_function(self,iterator):
        """Function for producing system responses from
        simulated parameter arrays produced in signal"""
        # If some combination of 20% done running, checkpoint to console
        if iterator in self.milestones:
            self.serialized_checkpoint(iterator)

        # Create first PRBS outside of loop
        u = self.uArray[iterator,:,:]
```

```
        # The transfer function from the 2nd input to the 1st output is
        # (3z + 4) / (6z^2 + 5z + 4).
        # num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
        # Iterate over each of the output dimensions and
        # add to numerator
        allY=np.zeros((self.nstep,self.outDim))
        for j in range(0,self.outDim):
            # Iterate over each of the input dimensions
            # and add to the numerator array
            numTemp = [[self.b_real_params[iterator,self.outDim*j+i]] for i in r
            denTemp = [[1.,-self.a_real_params[iterator,self.outDim*j+i]] for i
            kVals = [self.k_real_params[iterator,self.outDim*j+i] for i in range

            bigU=np.transpose(u)
            uSim=np.zeros_like(bigU)
            for (idx,row) in enumerate(bigU):
                try:
                    uSim[idx,:] = np.concatenate((np.zeros(kVals[idx]),row[:-kVa
                except:
                    uSim[idx,:] = row
            numTemp=np.array([numTemp]);denTemp=np.array([denTemp])
            sys = control.tf(numTemp,denTemp,1)
            _,y,_ = control.forced_response(sys,U=uSim,T=self.t)

            steps = int(self.nstep/self.length)
            l=self.length

            if self.disturbance:
                disturbed = self.add_disturbance()
            else:
                disturbed = np.zeros_like(y)

            for step in range(0,steps):
                allY[l*step:l*(step+1),j] = self.gauss_noise(y[l*step:l*(step+1)

            if self.disturbance:
                amax = np.max(allY)/np.max(disturbed)
                ratio = .25*amax
                disturbed = disturbed*ratio
                allY = allY + disturbed

        # Colors for plotting input/output signals properly
        colors = ['midnightblue','gray','darkgreen','crimson','olive','navy','li
                  'coral','darkorange','navy','r']

        '''
        if iterator<10:
            fig, axes = plt.subplots(3, 1,figsize=(15,5),dpi=400)
            plt.figure(figsize=(10,5),dpi=200)
            label1 = '$u_' + str(1) + '(t)$'
```

```
            label2 = '$w_' + str(1) + '(t)$'
            label3 = '$x_' + str(1) + '(t)$'
            label4 = '$y_' + str(1) + '(t)$'
            #plt.plot(self.t,self.uArray[iterator,:,:],colors[0],label=label1)
            #plt.plot(self.t,self.uArray[self.randint,:,:],colors[1],label=label
            axes[1].plot(self.t,disturbed,'navy',label=label2)
            axes[1].set(ylabel='w(t)')
            axes[0].plot(self.t,allY-disturbed,'red',label=label3)
            axes[0].set(ylabel='x(t)')
            axes[2].plot(self.t,allY,'purple',label=label4)
            #plt.ylabel("Measured Signal (5% Noise)")
            axes[2].set(ylabel='y(t)',xlabel='Time (s)')
        '''
        return allY


    def sys_simulation(self,stdev=5,b_possible_values=[.01,.99],
                       a_possible_values=[.01,.99],k_possible_values=[1,10],
                       order=False,disturbance=True,not_prbs=False):
        """
        Module which produces simulation of SISO/MIMO system given the input par
        Contains a loop which iterates for the total number of samples and appen
        to an array.

        Uses pseudo-random binary signal with amplitude in [-1,1] and linear fir
        order plus dead time system, modelled using transfer function class in
        the Control package to simulate the linear system.

        Purpose is to produce simulated system responses with varying quantities
        of noise to simulate real linear system responses in order to train and
        validate models built with the Model class.

        Parameters
        ----------
        inDim : int, default=2
            Number of input variables to MIMO system. Currently only set up
            to handle MIMO system with 2 inputs and 2 outputs.

        outDim : int, default=2
            Number of output variables from MIMO system. Currently only
            configured to handle MIMO with 2 inputs and 2 outputs.

        stdev : float, default=5.
            Standard deviation of gaussian error added to the simulated system.

        b_possible_values : tuple, default=(1,10)
            Possible range for gains. An equally spaced array between the maximu
            and minimum are chosen based on the number of simulations.

        a_possible_values : tuple, default=(1,10)
```

```
    Possible range for time constant. An equally spaced array between th
    maximum and minimum are chosen based on the number of simulations.

"""
# Access all the attributes from initialization
numTrials=self.numTrials; nstep=self.nstep; self.stdev = stdev
timelength=self.timelength; trainFrac=self.trainFrac
self.b_possible_values = b_possible_values; self.a_possible_values = a_p
self.k_possible_values = k_possible_values; self.order = order
self.disturbance=disturbance ; self.not_prbs = not_prbs

self.milestones = []
# Loop to create milestones to checkpoint data creation
for it in [2,4,6,8]:
   self.milestones.append(int((it/10)*numTrials))

# Set the type of the simulation to inform the data split
self.type = "MIMO"

# Initialize the arrays which will store the simulation data
orderList = []

# Make arrays containing parameters a,b and k
b_params_sampled = np.linspace(b_possible_values[0],b_possible_values[1]
a_params_sampled = np.linspace(a_possible_values[0],a_possible_values[1]
k_params_sampled = np.arange(k_possible_values[0],k_possible_values[1])
self.t = np.linspace(0,timelength-1,nstep)

# Make random number arrays for parameters
b_rand_int = np.random.randint(0,nstep,(numTrials*self.outDim*self.inDim
a_rand_int = np.random.randint(0,nstep,(numTrials*self.outDim*self.inDim
k_rand_int = np.random.randint(0,len(k_params_sampled),(numTrials*self.o

# Create parameter arrays and reshape
self.b_real_params = np.array([b_params_sampled[i] for i in b_rand_int])
self.a_real_params = np.array([a_params_sampled[i] for i in a_rand_int])
self.k_real_params = np.array([k_params_sampled[i] for i in k_rand_int])


if self.inDim>1:
    self.uArray = np.zeros((self.numTrials,self.nstep,self.inDim))
    for trial in range(0,numTrials):
        if self.not_prbs:
            seq=self.random_process()
        else:
            seq = self.PRBS()
        seq[-300:] = 0
        for dim in range(0,self.inDim):
            dimseq = np.zeros(self.nstep)
            dimseq[dim*self.length:dim*self.length+self.length] = seq
```

```
            self.uArray[trial,:,dim] = dimseq
else:
    # Make uArray for all data
    self.uArray = np.zeros((self.numTrials,self.nstep,self.inDim))
    for trial in range(0,numTrials):
        if self.not_prbs:
            seq=self.random_process()
        else:
            seq = self.PRBS()
        for dim in range(0,self.inDim):
            dimseq = np.zeros(self.nstep)
            for i in range(0,int(self.nstep/self.length)):
                dimseq[i*self.length:(i*self.length+self.length)]=seq
            self.uArray[trial,:,dim] = dimseq
            seq = -seq[::-1]


# Iterate over each of the simulations and add
# to simulation arrays
iterator=range(numTrials)
self.step=False
if not(self.order):
    self.yArray = np.array(list(map(self.y_map_function,iterator)))
elif self.order>1:
    #self.b_real_params = np.zeros((numTrials,self.outDim*self.inDim*sel
    self.a_real_params = np.zeros((numTrials,self.outDim*self.inDim*self
    #self.k_real_params = np.array([k_params_sampled[i] for i in k_rand_
    self.ySteps = np.zeros((self.numTrials,self.nstep,self.inDim))
    self.yArray = np.array(list(map(self.y_map_higher_function,iterator)
else:
    self.ySteps = np.zeros((self.numTrials,self.nstep,self.inDim))
    self.yArray = np.array(list(map(self.y_map_higher_function,iterator)
# Colors for plotting input/output signals properly
colors = ['midnightblue','gray','darkgreen','crimson','olive','navy','li
          'coral','darkorange','navy','r']


# Only plot every 100 input signals
for outit in range(self.numPlots):
    plt.figure(figsize=(10,5),dpi=200)
    plt.grid()
    for it in range(self.uArray.shape[-1]):
        label1 = '$u_' + str(it+1) + '(t)$'
        label2 = '$y_' + str(it+1) + '(t)$'
        plt.plot(self.t,self.uArray[outit,:,it],colors[it],label=label1)
        plt.plot(self.t,self.yArray[outit,:,it],colors[self.inDim+it],la
    plt.ylabel("Measured Signal (5% Noise)")
    plt.xlabel("Time Step (s)")
    plt.legend()
    plt.show()
```

```
        print(len(self.yArray))
        # Randomly pick training and validation indices
        index = range(0,len(self.yArray)*self.inDim*self.outDim)
        if self.trainFrac!=1:
            train = random.sample(index,int(trainFrac*len(self.yArray)*self.inDi
            test = [item for item in list(index) if item not in train]
        else:
            train=range(0,len(self.yArray*self.inDim*self.outDim))
            test=[]


        # Make it so that any of these attributes can be accessed
        # without needing to return them all from the function
        self.aVals = self.a_real_params
        self.bVals = self.b_real_params
        self.kVals = self.k_real_params
        self.orderList = orderList
        self.train = train
        self.test = test
        #if self.numPlots>0:
            #self.plot_parameter_space(self.a_real_params,self.b_real_params,sel

        return self.uArray,self.yArray,self.a_real_params,self.b_real_params,sel

    def closed_loop_forced(self):
        nTotal = self.numTrials ; trainFrac = self.trainFrac

        directory = self.pd + '/MATLAB code/'
        csv_1 = directory + 'inputArray.csv'
        csv_2 = directory + 'outputArray.csv'
        inputData = np.transpose(np.genfromtxt(csv_1,delimiter=','))
        outputData = np.transpose(np.genfromtxt(csv_2,delimiter=','))

        a,b = inputData.shape
        self.uArray = inputData[:,1:].reshape(a,b-1,1)
        self.yArray = outputData[:,1:].reshape(a,b-1,1)
        self.a_real_params = np.linspace(0.01,0.99,nTotal).reshape(a,1)
        self.b_real_params = np.linspace(0.01,0.99,nTotal).reshape(a,1)
        self.k_real_params = np.array([i%9 for i in range(0,100000)]).reshape(a,

        # Randomly pick training and validation indices
        index = range(0,len(self.yArray)*self.inDim*self.outDim)
        if self.trainFrac!=1:
            train = random.sample(index,int(trainFrac*len(self.yArray)*self.inDi
            test = [item for item in list(index) if item not in train]
        else:
            train=range(0,len(self.yArray*self.inDim*self.outDim))
            test=[]
```

```
        # Make it so that any of these attributes can be accessed
        # without needing to return them all from the function
        self.aVals = self.a_real_params
        self.bVals = self.b_real_params
        self.kVals = self.k_real_params
        self.train = train
        self.test = test

        return self.uArray,self.yArray,self.a_real_params,self.b_real_params,sel

    def preprocess(self,xData,yData,mutDim=1):
        """This function uses the training and testing indices produced during
        simulate() to segregate the training and validation sets"""
        # If array has more than 2 dimensions, use
        # axis=2 when reshaping, otherwise set to 1
        try:
            numTrials,_,numDim= xData.shape
        except:
            numTrials,_ = xData.shape
            numDim=1

        # Select training and validation data based on training
        # and testing indices set during simulation
        trainspace = xData[self.train]
        valspace = xData[self.test]

        x_train= trainspace.reshape((math.floor(numTrials*self.trainFrac),self.l
        x_val = valspace.reshape((math.floor(numTrials*(1-self.trainFrac)),self.

        try:
            y_val = np.array([yData[i,:] for i in self.test])
            y_train = np.array([yData[i,:] for i in self.train])
        except:
            y_val = np.array([yData[i] for i in self.test])
            y_train = np.array([yData[i] for i in self.train])

        if self.trim>0:
            return x_train[:,:-self.trim,:],x_val[:,:-self.trim,:],y_train,y_val
        else:
            return x_train,x_val,y_train,y_val,numDim

    def stretch_MIMO(self,name):
        """Function that takes the input parameters and stacks into one
        array, then processes so that data can be used for any size
        MIMO system. Not used if SISO system"""
        bVals=self.bVals; aVals=self.aVals; kVals=self.kVals
        uArray=self.uArray; yArray=self.yArray
        a,b,c = np.shape(self.uArray)
        self.xDataMat = np.full((a*self.outDim,b,c),0.)
        self.yDataMat = np.full((a*self.outDim,self.inDim),0.)
```

```
        if name=='b':
            for j in range(0,self.inDim):
                dim = self.inDim
                self.yDataMat[a*j:a*(j+1),:] = bVals[:,dim*j:dim*(j+1)]

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*i:a*(i+1),:,j] = yArray[:,:,i] * uArray[:,:,
        elif name=='a':
            for j in range(0,self.inDim):
                dim = self.inDim
                self.yDataMat[a*j:a*(j+1),:] = aVals[:,dim*j:dim*(j+1)]

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*i:a*(i+1),:,j] = yArray[:,:,i] - uArray[:,:,
        else:
            for j in range(0,self.inDim):
                dim = self.inDim
                self.yDataMat[a*j:a*(j+1),:] = kVals[:,dim*j:dim*(j+1)]

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*i:a*(i+1),:,j] = yArray[:,:,i] - uArray[:,:,

        print(self.xDataMat.shape)
        return self.xDataMat,self.yDataMat

    def stretch_MIMO_multi(self,name):
        """Function that takes the input parameters and stacks into one
        array, then processes so that data can be used for any size
        MIMO system. Not used if SISO system"""
        bVals=self.bVals; aVals=self.aVals; kVals=self.kVals
        uArray=self.uArray; yArray=self.yArray
        a,b,c = np.shape(self.uArray)
        self.xDataMat = np.full((a*self.outDim*self.inDim,self.length),0.)
        self.yDataMat = np.full((a*self.outDim*self.inDim,2),0.)
        k = self.length

        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                dim=self.inDim
                self.yDataMat[a*(dim*i+j):a*(dim*i+j+1),0] = aVals[:,(dim*i+j):(
                self.yDataMat[a*(dim*i+j):a*(dim*i+j+1),1] = bVals[:,(dim*i+j):(

        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),...] = yAr

        return self.xDataMat,self.yDataMat
```

```
def stretch_MIMO_sequential(self,name):
    """Function that takes the input parameters and stacks into one
    array, then processes so that data can be used for any size
    MIMO system. Not used if SISO system"""
    bVals=self.bVals; aVals=self.aVals; kVals=self.kVals
    uArray=self.uArray; yArray=self.yArray
    a,b,c = np.shape(self.uArray)
    self.xDataMat = np.full((a*self.outDim*self.inDim,self.length),0.)
    self.yDataMat = np.full((a*self.outDim*self.inDim),0.)
    k = self.length
    if name=='b':
        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                dim=self.inDim
                self.yDataMat[a*(dim*i+j):a*(dim*i+j+1)] = bVals[:,(dim*i+j)

        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),...] =

    elif name=='a':
        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                dim=self.inDim
                self.yDataMat[a*(dim*i+j):a*(dim*i+j+1)] = aVals[:,(dim*i+j)

        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),...] =
    else:
        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                dim=self.inDim
                self.yDataMat[a*(dim*i+j):a*(dim*i+j+1)] = kVals[:,(dim*i+j)

        for i in range(0,self.outDim):
            for j in range(0,self.inDim):
                self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),...] =

    return self.xDataMat,self.yDataMat

def stretch_convolutional(self,name):
    """Function that takes the input parameters and stacks into one
    array, then processes so that data can be used for any size
    MIMO system. Not used if SISO system"""
    bVals=self.bVals; aVals=self.aVals; kVals=self.kVals
    uArray=self.uArray; yArray=self.yArray
    a,b,c = np.shape(self.uArray)
    self.xDataMat = np.full((a*self.outDim*self.inDim,self.length,5),0.)
```

```
        self.yDataMat = np.full((a*self.outDim*self.inDim,2),0.)
        k = self.length
        if name=='b':
            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    dim=self.inDim
                    self.yDataMat[a*(dim*i+j):a*(dim*i+j+1),0] = bVals[:,(dim*i+
                    self.yDataMat[a*(dim*i+j):a*(dim*i+j+1),1] = aVals[:,(dim*i+

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,0] =
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,1] =
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,2] =
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,3] =
                    for zz in range(0,a):
                        self.xDataMat[zz,:,4] = np.convolve(yArray[zz,j*k:(j+1)*

        elif name=='a':
            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    dim=self.inDim
                    self.yDataMat[a*(dim*i+j):a*(dim*i+j+1)] = aVals[:,(dim*i+j)

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,0] =
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,1] =
        else:
            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    dim=self.inDim
                    self.yDataMat[a*(dim*i+j):a*(dim*i+j+1)] = kVals[:,(dim*i+j)

            for i in range(0,self.outDim):
                for j in range(0,self.inDim):
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,0] =
                    self.xDataMat[a*(self.inDim*i+j):a*(self.inDim*i+j+1),:,1] =

        return self.xDataMat,self.yDataMat


    def system_validation(self,b_possible_values=[.01,.99],a_possible_values=[.0
                          k_possible_values=[1,10],order=False,disturbance=False
        """This function makes it easier to run a bunch of simulations and
        automatically return the validation and testing sets without
        calling each function separately. """
        # Since no training is occurring, can skip separation of testing and val
        self.trainFrac = 1
```

```
        uArray,yArray,aVals,bVals,kVals,train,test = self.sys_simulation(stdev=s
                    b_possible_values=b_possible_values,a_possible_values=a_poss
                    k_possible_values=k_possible_values,order=order,disturbance=

        a,b,c = np.shape(uArray)

        self.xData ={};
        self.yData={}
        self.names = ["b","a","k"]

        for (i,name) in enumerate(self.names):
            # Develop separate model for each output variable
            x,y = self.stretch_MIMO_sequential(name)

            a,b = x.shape
            x = x.reshape((a,b,1))

            self.xData[name] = x
            self.yData[name] = y

        return self.xData,self.yData

    def system_validation_multi(self,b_possible_values=[.01,.99],
                                a_possible_values=[.01,.99],k_possible_values=[0
                                order=False,disturbance=False,not_prbs=False):
        """This function makes it easier to run a bunch of simulations and
        automatically return the validation and testing sets without
        calling each function separately. """
        # Since no training is occurring, can skip separation of testing and val
        self.trainFrac = 1

        uArray,yArray,aVals,bVals,kVals,train,test = self.sys_simulation(stdev=s
                    b_possible_values=b_possible_values,a_possible_values=a_poss
                    k_possible_values=k_possible_values,order=order,disturbance=

        a,b,c = np.shape(uArray)

        self.xData ={};
        self.yData={}
        self.names = ["b","a"]

        for (i,name) in enumerate(self.names):
            # Develop separate model for each output variable
            x,y = self.stretch_MIMO_multi(name)

            a,b = x.shape
            x = x.reshape((a,b,1))

            self.xData[name] = x
            self.yData[name] = y
```

```
        return self.xData,self.yData

    def closed_loop_validation(self,b_possible_values=[.01,.99],
                               a_possible_values=[.01,.99],k_possible_values=[1,
        """This function makes it easier to run a bunch of simulations and
        automatically return the validation and testing sets without
        calling each function separately. """
        # Since no training is occurring, can skip separation of testing and val
        self.trainFrac = 1

        uArray,yArray,aVals,bVals,kVals = self.closed_loop_forced()

        a,b,c = np.shape(uArray)

        self.xData ={};
        self.yData={}
        self.names = ["b","a","k"]

        for (i,name) in enumerate(self.names):
            # Develop separate model for each output variable
            x,y = self.stretch_MIMO_sequential(name)

            a,b = x.shape
            x = x.reshape((a,b,1))

            self.xData[name] = x
            self.yData[name] = y

        return self.xData,self.yData



    def y_map_higher_function(self,iterator):
        """This function simulates higher order systems in order to
        validate the model as a predictor of higher order systems.
        Cannot be used to generate signals for training"""
        # If some combination of 20% done running, checkpoint to console
        if iterator in self.milestones:
            self.serialized_checkpoint(iterator)

        # Create first PRBS outside of loop
        u = self.uArray[iterator,:,:]
        a,b = self.b_real_params.shape
        self.bFinal = np.zeros((a,b,self.order))
        self.aFinal = np.zeros((a,b,self.order+1))
        if not(hasattr(self, 'test_sequence')):
            self.test_sequence=self.PRBS()

        # The transfer function from the 2nd input to the 1st output is
```

```
# (3s + 4) / (6s^2 + 5s + 4).
# num = [[[1., 2.], [3., 4.]], [[5., 6.], [7., 8.]]]
# Iterate over each of the output dimensions and
# add to numerator
allY=np.zeros((self.nstep,self.outDim))
stepY=np.zeros((self.nstep,self.outDim))
for j in range(0,self.outDim):
    # Iterate over each of the input dimensions
    # and add to the numerator array
    num=[];denom=[]
    for i in range(self.inDim):
        b = self.b_real_params[iterator,self.outDim*j+i]
        a = self.a_real_params[iterator,self.outDim*j+i]
        numTemp=np.zeros(self.order);denTemp=np.zeros(self.order+1)
        denTemp[0]=1.
        for o in range(1,self.order+1):
            if self.order==1:
                numo = b
                deno = -a
                numTemp[0]=numo
                denTemp[1]=deno
            else:
                deno = np.random.uniform(0.2,self.a_possible_values[1]/2
                denTemp[o] = -deno

                while np.sum(abs(denTemp[1:]))>=1:
                    deno=deno/2
                    denTemp[o]=deno

                self.a_real_params[iterator,self.inDim*self.order*j+i*se

            #denTemp.append(-deno)
            #mean = abs(np.sum(denTemp[1:]))

        num.append([b])
        denom.append(denTemp)
    kVals = [self.k_real_params[iterator,self.outDim*j+i] for i in range

    bigU=np.transpose(u)
    uSim=np.zeros_like(bigU)
    for (idx,row) in enumerate(bigU):
        uSim[idx,:] = np.concatenate((np.zeros(kVals[idx]),row[:-kVals[i
    num=np.array([num]);denom=np.array([denom])
    sys = control.tf(num,denom,1)
    _,y,_ = control.forced_response(sys,U=uSim,T=self.t)
    _,sY,_ = control.forced_response(sys,U=self.test_sequence,T=np.linsp

    allY[:,j] = self.gauss_noise(y,self.stdev)
    try:
        stepY[:,j] = self.gauss_noise(sY,self.STDEVS[-1])
```

```
        except:
            stepY[:,j] = self.gauss_noise(sY,self.stdev)
        for idx in range(stepY.shape[-1]):
            row = stepY[...,i]
            temp=np.zeros_like(row)
            temp[kVals[i]:]= row[:-kVals[i]]
            temp[:kVals[i]] = 0
            stepY[...,i] = temp
    self.ySteps[iterator,...]=stepY
    return allY
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 30 17:32:17 2020

@author: RileyBallachay
"""
import os
from os import path
from pathlib import Path
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_probability as tfp
tfd = tfp.distributions
tfk = tf.keras
tfb = tfp.bijectors
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler
from tkinter import Tk
from tkinter.filedialog import askdirectory
import matplotlib.pyplot as plt
from Signal import Signal
from tensorflow.keras import backend as K
from tensorflow.keras import losses
import datetime
import control as control
import seaborn as sns

class MyThresholdCallback(tf.keras.callbacks.Callback):
    """
    Callback class to stop training model when a target
    coefficient of determination is achieved on validation set.
    """

    def __init__(self, threshold):
        super(MyThresholdCallback, self).__init__()
        self.threshold = threshold

    def on_epoch_end(self, epoch, logs=None):
        val_acc = logs["val_coeff_determination"]
        if val_acc >= self.threshold:
            self.model.stop_training = True

class Model:
    """
    Class for fitting and predicting first order state space
    models. All models are built to be used on pseudo-binary
```

```
(aka step-input) data.

Builds models using keras with tensorflow backend. Currently
only works for SISO systems. Will work with MIMO in the future.

The Purpose of this class is to fit k, a and b for first
order systems using data simulated in the Signal class. Models are
stored using the h5 format and can be used to estimate accuracy.

Parameters
----------
Modeltype : string, default='regular'
    Two types of models exist: regular and probability. Probability
    model uses lamba function embedded at the final layer of keras
    network to estimate mean and standard deviation for regression.
    Regular model is built using keras Dense output and probability
    is estimated based on uncertainty in training data with a set
    gaussian noise with standard deviation stdev, set as a parameter
    in the call to Signal.

nstep : int, default=100
    Parameter shared with the signal class. Will inherit in the future.

"""

def __init__(self,Modeltype='probability',sig=False):
    self.Modeltype = Modeltype
    self.names = ["b","a","k"]
    self.modelDict = {}
    self.special_value = -99
    self.cwd = str(os.getcwd())
    self.pd = str(Path(os.getcwd()).parent)

def load_model(self,sig,directory=False,check=False):
    """Loads one of two first order models: probability or regular. Iterates
    over directory and loads alphabetically. If more than 3 models exist in
    directory, it will load them indiscriminately."""
    modelList = []
    self.inDim = sig.inDim; self.outDim = sig.outDim
    self.nstep = sig.nstep; self.trainFrac = sig.trainFrac
    self.length,self.width,self.height=sig.xData['b'].shape
    if not(directory):
        loadDir =  askdirectory(title='Select Folder With Trained Model')
    else:
        loadDir = directory

    models = ['b.cptk','a.cptk','k.cptk']
    for filename in models:
        if filename=='.DS_Store':
            continue
```

```
            print(filename)

            if not(check):
                model = self.mutable_model()
            else:
                model = self.mutable_model()

            model.load_weights(loadDir+filename)
            modelList.append(model)
        for i in range(0,3):
            self.modelDict[self.names[i]] = modelList[i]


    def load_and_train(self,sig,epochs=50,probabilistic=True,saveModel=True,
                       batchSize=16,plotLoss=False,plotVal=False):
        """Calls load_SISO and continues training model, saving the updated
        model to a new folder to avoid overwriting"""

        self.train_model(sig, saveModel=saveModel,epochs=epochs,
                         checkpoint=True,batchSize=batchSize)


    def train_model(self,sig=False,plotLoss=True,plotVal=True,epochs=50,saveMode
                    checkpoint=False,batchSize=16,trainConvolutional=False):
        """
        Takes Signal object with input and output data, separates data in traini
        and validation sets, transform data for neural network and uses training
        set to produce neural network of specified architecture. Works for MIMO.

        Separate models are produced for each output variable. Systems are
        considered to behave as linear combinations of linear systems.
            b: System response (y) and b used to construct model
            a: System response (y) and a used to construct model

        Parameters
        ----------
        sig: Signal (object), default=False
            Must provide instance of signal class in order to build model or use
            for prediction. Else, will fail to train.

        plotLoss: bool, default=True
            Plot training and validation loss after training finishes. Saves to
            a folder with the date and time if saveModel is set to true.

        plotVal: bool, default=True
            Plots validation data with coefficient of determination after traini
            model. Saves to a folder with the date and time if saveModel is
            set to true.

        probabilistic: bool, default=True
```

```
        Choice between probabilistic model (calls FOPTD_probabilistic) and
        regular (FOPTD). If regular, prediction will include uncertainties
        built in with the validation set.

    saveModel: bool, default=True
        Decide whether or not to save the models from training
    """
    parameters = ['b','a','k']
    self.numBatches  = int(sig.numTrials*0.7/batchSize)
    # You have to construct a signal with all the necessary parameters befor
    if not(sig):
        print("Please initialize the class signal with model parameters firs
        return

    # If the loss and accuracy plots are gonna be saved, saveModel must = Tr
    parentDir = self.pd+"/Models/"
    time = str(datetime.datetime.now())[:16]
    plotDir = parentDir + time + '/'
    checkptDir = plotDir + 'Checkpoints/'
    os.mkdir(plotDir)
    os.mkdir(checkptDir)

    # iterate over each of the parameters, train the model, save to the mode
    # path and plot loss and validation curves
    modelList = []
    first=True

    # Iterate over b and a parameters
    for (k,parameter) in enumerate(parameters):
        if not(trainConvolutional):
            xData,yData = sig.stretch_MIMO_multi(parameter)
            #colors = ['olive','navy','lightcoral','indigo']
            #plt.figure(k,dpi=300,figsize=(10,5))
            #for jj in range(0,10):
            #    label1 = '$y_' + str(jj+1) + '(t)$' +'-'+'$u_' + str(1) + '
            #    plt.plot(xData[jj*sig.numTrials*sig.outDim,:-300],colors[jj

            #plt.xlabel('Time Step (s)')
            #plt.ylabel('Signal Difference')
            #plt.legend()

            x_train,x_val,y_train,y_val,numDim = sig.preprocess(xData,yData)

        else:
            xData,yData = sig.stretch_convolutional(parameter)
            x_train,x_val,y_train,y_val,numDim = sig.preprocess(xData,yData,

        self.outDim = sig.outDim;self.inDim=sig.inDim
        self.length,self.width,self.height=x_train.shape
        self.numTrials=sig.numTrials;self.nstep=sig.nstep
```

```
self.maxLen = sig.maxLen
self.trainFrac = sig.trainFrac

"""
# Check the dimension of data to ensure that an architecture
# exists for the shape of the data. If not, then it will
# prompt the user to make a new architecture
if checkpoint:
    if first:
        self.x_train = x_train; self.y_train = y_train
        valPath = '/Users/RileyBallachay/Documents/Fifth Year
        /RNNSystemIdentification/Model Validation/'
        name ='MIMO ' + str(sig.inDim) + 'x' + str(sig.outDim)
        path = valPath + name + '/Checkpoints/'
        self.load_model(sig,directory=path,check=True)
        first = False

    model = self.modelDict[parameter]
else:
"""
if not(trainConvolutional):
    model = self.mutable_model()
else:
    model = self.mutable_convolutional()

print(model.summary())

# Set threshold to stop training when coefficient of determinatio ge
# Section for including callbacks (custom and checkpoint)
checkpoint_path = checkptDir + self.names[k] + '.cptk'
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint
            save_weights_only=True,save_best_only=True,verbose=1)
MTC = MyThresholdCallback(0.995)

print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=batchSize,
    epochs=epochs,
    # We pass some validation for
    # monitoring validation loss and metrics
    # at the end of each epoch
    validation_data=(x_val, y_val),
    callbacks=[MTC,cp_callback]
)

# Save the training and validation loss to a text file
numpy_loss_history = np.array(history.history['loss'])
numpy_val_history = np.array(history.history['val_loss'])
```

```
np.savetxt(plotDir+parameter+'_loss.txt', numpy_loss_history, delimi
np.savetxt(plotDir+parameter+'_val_loss.txt', numpy_val_history, del

# Store each model in dictionary and list
#modelList.append(model)
#self.modelDict[self.names[k]] = model
#yhat = model(x_val)
#predictions = np.array(yhat.mean())
#stddevs = np.array(yhat.stddev())

# Only save model if true
"""
if saveModel:
    modelpath = plotDir+self.names[k]
    fileOverwriter=0
    while os.path.isfile(modelpath):
        modelpath = plotDir+self.names[k]+"_"+str(fileOverwriter)
        fileOverwriter+=1
    model.save(modelpath)

# Plot learning curve for each parameter
if plotLoss:
    plt.figure(dpi=200)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss for '+ parameter)
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    if saveModel:
        plt.savefig(plotDir+parameter+'_loss'+'.png')
    plt.show()
"""
# Plot predicted and actual value for each paramter and
# coefficient of determination
"""
if plotVal:
    fig, axes = plt.subplots(1, sig.inDim,dpi=200)
    fig.add_subplot(111, frameon=False)
    plt.tick_params(labelcolor='none', top=False,
                    bottom=False, left=False, right=False)
    for (i,ax) in enumerate(axes):
        ax.plot(y_val[:,i],predictions[:,i],'.b',label=parameter)
        print(stddevs.shape)
        r2 =("r\u00b2 = %.3f" % r2_score(y_val[:,i],predictions[:,i]
        ax.errorbar(y_val[:,i],predictions[:,i],yerr=stddevs[:,i]*2,
                    fmt='none',ecolor='green')
        ax.plot(np.linspace(0,10),np.linspace(0,10),'--r',label = r2
        ax.legend(prop={'size': 8})
```

```
            plt.ylabel("Predicted Value of "+ parameter)
            plt.xlabel("True Value of "+ parameter)

            if saveModel:
                plt.savefig(plotDir+self.names[k]+'.png')
            plt.show()
        """
    return


def MSE(self,y_true,y_pred):
    """Calculate mean squared error of
    predicted and actual system response."""
    mse = sum((y_true-y_pred)**2)
    return mse

def predict_system(self,sig,plotPredict=True,savePredict=False,probabilityCa
                   stepResponse=True):
    """
    Takes Signal object with input and output data, uses stored model
    to predict parameters based off simulated data, then plots
    system output based on predicted parameters.

    Plots response using predicted parameters along with real response,
    as well as real and predicted parameters.

    Parameters
    ----------
    sig : Signal (object), default=False
        Must provide instance of signal class in order to build model or use
        for prediction. Else, will fail to train.

    plotPredict : bool, default=True
        Plots predicted response with real response.

    savePredict : bool,default=False
        Determine whether or not to save predicted responses.

    probabilityCall : bool, default=False
        Included to avoid building probability object while trying to
        build probability object, leading to endless loop.
    """
    if not(isinstance(sig,Signal)):
        print("You need to predict with an instance of signal!")
        return

    # Model is used to predict all out the output variables linearly,
    # so the array is cut depending on which transfer function parameters
    # correspond to each output variable, then stacked as new trials
    k_yhat = self.modelDict['k'](sig.xData['k'])
```

```
b_yhat = self.modelDict['b'](sig.xData['b'])
a_yhat = self.modelDict['a'](sig.xData['a'])

predDict = dict()
errDict = dict()
self.stdDict = dict()
predDict['b'] = np.array(b_yhat.mean()).flatten()
errDict['b'] = np.array(2*b_yhat.stddev()).flatten()
self.stdDict['b'] = np.mean(errDict['b'])

predDict['a'] = np.array(a_yhat.mean()).flatten()
errDict['a'] = np.array(2*a_yhat.stddev()).flatten()
self.stdDict['a'] = np.mean(errDict['a'])

predDict['k'] = np.array(k_yhat.mean()).flatten()
errDict['k'] = np.array(2*k_yhat.stddev()).flatten()
self.stdDict['k'] = np.mean(errDict['k'])

nameDict = dict()
nameDict['b'] = 'b'
nameDict['a'] = 'a'
nameDict['k'] = 'k'
sns.set_style("dark")

if stepResponse and sig.inDim==1:
    self.step_response_validation(sig,predDict,errDict)
else:
    fig, axes = plt.subplots(1, 3,figsize=(15,5),dpi=400)
    for (idx,parameter) in enumerate(['a','b','k']):
        sig.yData[parameter] = sig.yData[parameter].flatten()
        sig.xData[parameter] = sig.xData[parameter].flatten()
        ax = axes[idx]
        fig.add_subplot(111, frameon=False)
        plt.tick_params(labelcolor='none', top=False, bottom=False, left
        ax.plot(sig.yData[parameter],predDict[parameter],'.k',label=name
        r2 =("r\u00b2 = %.3f" % r2_score(sig.yData[parameter],predDict[p
        ax.errorbar(sig.yData[parameter],predDict[parameter],
                    yerr=errDict[parameter],fmt='none',ecolor='green',
                    label=('Avg. Unc.=%.2f' %self.stdDict[parameter]))
        if parameter=='k':
            ax.plot(np.linspace(0,10),np.linspace(0,10),'--r',label = r2
        else:
            ax.plot(np.linspace(0,1),np.linspace(0,1),'--r',label = r2)
        ax.legend()

    for ax in axes.flat:
        ax.label_outer()
        ax.xaxis.grid()
        ax.yaxis.grid()
```

```
            plt.ylabel("Predicted Parameter Value")
            plt.xlabel("True Parameter Value")
        return predDict, errDict

    def predict_multinomial(self,sig,plotPredict=True,savePredict=False,probabil
                            stepResponse=True):
        """
        Takes Signal object with input and output data, uses stored model
        to predict parameters based off simulated data, then plots
        system output based on predicted parameters.

        Plots response using predicted parameters along with real response,
        as well as real and predicted parameters.

        Parameters
        ----------
        sig : Signal (object), default=False
            Must provide instance of signal class in order to build model or use
            for prediction. Else, will fail to train.

        plotPredict : bool, default=True
            Plots predicted response with real response.

        savePredict : bool,default=False
            Determine whether or not to save predicted responses.

        probabilityCall : bool, default=False
            Included to avoid building probability object while trying to
            build probability object, leading to endless loop.
        """
        if not(isinstance(sig,Signal)):
            print("You need to predict with an instance of signal!")
            return

        # Model is used to predict all out the output variables linearly,
        # so the array is cut depending on which transfer function parameters
        # correspond to each output variable, then stacked as new trials
        a_yhat = self.modelDict['a'](sig.xData['a'])

        predDict = dict()
        errDict = dict()
        self.stdDict = dict()

        print(a_yhat.covariance)
        predDict['a'] = np.array(a_yhat.mean())[:,0]
        errDict['a'] = np.array(2*a_yhat.stddev())[:,0]
        self.stdDict['a'] = np.mean(errDict['a'])

        predDict['b'] = np.array(a_yhat.mean())[:,1]
        errDict['b'] = np.array(2*a_yhat.stddev())[:,1]
```

```
        self.stdDict['b'] = np.mean(errDict['b'])

        nameDict = dict()
        nameDict['a'] = 'a'
        nameDict['b'] = 'b'
        sns.set_style("dark")

        tempdict = dict()
        tempdict['a'] = sig.yData['a'][:,0]
        tempdict['b'] = sig.yData['a'][:,1]

        if stepResponse and sig.inDim==1:
            self.step_response_validation(sig,predDict,errDict)
        else:
            fig, axes = plt.subplots(1, 2,figsize=(12,5),dpi=400)
            for (idx,parameter) in enumerate(['a','b']):
                ax = axes[idx]
                fig.add_subplot(111, frameon=False)
                plt.tick_params(labelcolor='none', top=False, bottom=False, left
                ax.plot(tempdict[parameter],predDict[parameter],'.k',label=nameD
                r2 =("r\u00b2 = %.3f" % r2_score(tempdict[parameter],predDict[pa
                ax.errorbar(tempdict[parameter],predDict[parameter],
                            yerr=errDict[parameter],fmt='none',ecolor='green',
                            label=('Avg. Unc.=%.2f' %self.stdDict[parameter]))
                ax.plot(np.linspace(0,1),np.linspace(0,1),'--r',label = r2)
                ax.legend()

            for ax in axes.flat:
                ax.label_outer()
                ax.xaxis.grid()
                ax.yaxis.grid()

            plt.ylabel("Predicted Parameter Value")
            plt.xlabel("True Parameter Value")
        self.results = dict()
        self.results['a'] = predDict['a'][:len(predDict['a'])//2]
        self.results['b'] = predDict['a'][len(predDict['a'])//2:]
        return predDict, errDict

    def step_response_validation(self,sig,predDict,errDict):
        """Pick random parameterizations from the gaussian
        distribution of each parameter and plot. Finish by
        plotting the mean response and labeling with SSE"""
        sse_mean = np.zeros(sig.numTrials)
        sse_variance = np.zeros(sig.numTrials)
        sses = np.zeros(99)
        for i in range(0,len(predDict['b'])):
            if i<10:
                plt.figure(i+100,dpi=200)
            for j in range(0,100):
```

```
            if j<99:
                gauss_b = np.random.normal(predDict['b'][i],errDict['b'][i]/
                gauss_a = np.random.normal(predDict['a'][i],errDict['a'][i]/
                gauss_k = abs(int(np.ceil(np.random.normal(predDict['k'][i],
                if gauss_k==0:
                    gauss_k=1
                numTemp = [gauss_b]
                denTemp = [1.,-gauss_a]
                sys = control.tf(numTemp,denTemp,1)
                _,resp,_ = control.forced_response(sys,U=sig.test_sequence,
                                                   T=np.linspace(0,sig.lengt
                temp = np.zeros_like(resp)
                temp[gauss_k:] = resp[:-gauss_k]
                if i<10:
                    plt.plot(sig.t[:200],temp[:200]/np.max(sig.ySteps[i]),'m
                tempo = sig.ySteps[i][:]
                sses[j] = self.sum_error(tempo,temp)

            else:
                numTemp = [predDict['b'][i]]
                denTemp = [1.,-predDict['a'][i]]
                sys = control.tf(numTemp,denTemp,1)
                _,resp,_ = control.forced_response(sys,U=sig.test_sequence,
                                                   T=np.linspace(0,sig.lengt
                temp = np.zeros_like(resp)
                theta=abs(int(np.ceil(predDict['k'][i])))
                if theta==0:
                    theta=1
                temp[theta:] = resp[:-theta]
                tempo = sig.ySteps[i][:]
                if i<10:
                    plt.plot(sig.t[:200],sig.ySteps[i][:200]/np.max(sig.ySte
                             'g--',label='True Response, STDEV=%.1f'%sig.STD
                sse = self.sum_error(tempo,temp)
                sse_mean[i] = sse
                sse_variance[i] = np.var(sses)
                if i<10:
                    plt.plot(sig.t[:200],temp[:200]/np.max(sig.ySteps[i]),
                             'r',label='Estimated Response, SSE=%.2f'%sse,li
                    plt.legend()

        df = pd.DataFrame()
        df['Gauss'] = sig.STDEVS
        df['Mean SSE'] = sse_mean
        df['Var SSE'] = sse_variance
        df['order'] = np.full(len(sse_mean),sig.order)
        df['b'] = sig.yData['b'].flatten()
        df['a'] = sig.yData['a'].flatten()
        df['k'] = sig.yData['k'].flatten()
        df['b Pred'] = predDict['b']
```

```
        df['a Pred'] = predDict['a']
        df['k Pred'] = predDict['k']
        df['b Std'] = errDict['b']
        df['a Std'] = errDict['a']
        df['k Std'] = errDict['k']

        path = '/Users/RileyBallachay/Documents/Fifth Year/RNNSystemIdentificati
        file = 'step_data_1.csv'
        i=1
        while os.path.isfile(path+file):
            i+=1
            file = 'step_data_' + str(i) + '.csv'

        df.to_csv(path+file)
        return

    def sum_error(self,a,b):
        # Compute the sum of squared error between
        # the real and mean predicted response
        atemp=np.array(a[:100]).flatten()
        btemp=np.array(b[:100]).flatten()
        sse = ((atemp-btemp)**2).sum(axis=0)
        return sse


    def coeff_determination(y_true, y_pred):
        "Coefficient of determination for callback"
        SS_res =  K.sum(K.square( y_true[:,1]-y_pred[:,1] ))
        SS_tot = K.sum(K.square( y_true[:,1] - K.mean(y_true[:,1]) ) )
        return ( 1 - SS_res/(SS_tot + K.epsilon()) )

    # Specify the surrogate posterior over `keras.layers.Dense` `kernel` and `bi
    def posterior_mean_field(kernel_size, bias_size=0, dtype=None):
      n = kernel_size + bias_size
      c = np.log(np.expm1(1.))
      return tf.keras.Sequential([
          tfp.layers.VariableLayer(2 * n, dtype=dtype,initializer='glorot_unifor
          tfp.layers.DistributionLambda(lambda t: tfd.Independent(
              tfd.Normal(loc=t[..., :n],scale=1e-5 +
                         tf.nn.softplus(c + t[..., n:])),reinterpreted_batch_ndi

# Specify the prior over `keras.layers.Dense` `kernel` and `bias`.
    def prior_trainable(kernel_size, bias_size=0, dtype=None):
      n = kernel_size + bias_size
      return tf.keras.Sequential([
          tfp.layers.VariableLayer(n, dtype=dtype,initializer='glorot_uniform'),
          tfp.layers.DistributionLambda(lambda t: tfd.Independent(
              tfd.Normal(loc=t, scale=0.1),reinterpreted_batch_ndims=1)),])

    def customloss(self,y1,y2):
```

```
        rand = np.random.normal(-1,1)
        t = np.linspace(0,100,100)
        T = np.array([t,]*100)
        #y2 = y2.prob(y1)
        y_true = tf.map_fn(fn = lambda x: y1[:,0] *
                           tf.math.exp(-x/y1[:,1]),elems=tf.constant(T,dtype=tf.
        y_pred = tf.map_fn(fn = lambda x: y2[:,0] *
                           tf.math.exp(-x/y2[:,1]),elems=tf.constant(T,dtype=tf.
        return K.sum(K.square( y_true-y_pred ))

    def mutable_convolutional(self):
        "Probabilistic model for SISO data"
        #negloglik = lambda y, rv_y: -rv_y.log_prob(y[:])
        model = tf.keras.Sequential([
        tf.keras.layers.LSTM(100, activation='tanh', input_shape=(None, 5)),
        tf.keras.layers.Dense(100,activation='softplus'),
        tf.keras.layers.Dense(2,activation='softplus')])
        model.compile(optimizer='adam', loss=self.customloss,metrics=[Model.coef
        return model

    def mutable_model(self):
        "Probabilistic model for SISO data"
        negloglik = lambda y, rv_y: -rv_y.log_prob(y[:])
        model = tf.keras.Sequential([
        tf.keras.layers.LSTM(100, activation='tanh', input_shape=(None, 1)),
        tfp.layers.DenseVariational(4*1,Model.posterior_mean_field,Model.prior_t
        tfp.layers.DistributionLambda(lambda t: tfd.Normal(loc = t[..., :2],
        scale = (1e-3 + tf.math.softplus(0.1 * t[...,2:])),)),])
        model.compile(optimizer='adam', loss=negloglik,metrics=[Model.coeff_dete
        return model

    def mutable_model_nomulti(self):
        "Probabilistic model for SISO data"
        negloglik = lambda y, rv_y: -rv_y.log_prob(y[:])
        model = tf.keras.Sequential([
        tf.keras.layers.LSTM(100, activation='tanh', input_shape=(None, 1)),
        tfp.layers.DenseVariational(4*1,Model.posterior_mean_field,
                                    Model.prior_trainable,activation='linear',kl
        tfp.layers.DistributionLambda(lambda t: tfd.Normal(loc = t[..., :2],
        scale = (1e-3 + tf.math.softplus(0.1 * t[...,2:])),)),])
        model.compile(optimizer='adam', loss=negloglik,metrics=[Model.coeff_dete
        return model

    def mutable_model_safecopy(self):
        "Probabilistic model for SISO data"
        negloglik = lambda y, rv_y: -rv_y.log_prob(y[:])
        model = tf.keras.Sequential([
        tf.keras.layers.LSTM(100, activation='tanh', input_shape=(None, 1)),
        tf.keras.layers.Dense(100,activation='softplus'),
        tfp.layers.DenseVariational(2*1,Model.posterior_mean_field,
```

```
                                    Model.prior_trainable,activation='linear',kl
        tfp.layers.DistributionLambda(lambda t: tfd.Normal(loc = t[..., :1],
        scale = (1e-3 + tf.math.softplus(0.1 * t[...,1:])),)),])
        model.compile(optimizer='adam', loss=negloglik,metrics=[Model.coeff_dete
        return model

    def mutable_model_noAttribute(x_train,y_train):
        length,width,height = x_train.shape
        outheight,outwidth = y_train.shape
        "Probabilistic model for SISO data"
        negloglik = lambda y, rv_y: -rv_y.log_prob(y[:])
        model = tf.keras.Sequential([
        tf.keras.layers.LSTM(100, activation='tanh',input_shape=(None,height)),
        tfp.layers.DenseVariational(2*1,Model.posterior_mean_field,
                                    Model.prior_trainable,activation='linear',kl
        tfp.layers.DistributionLambda(lambda t: tfd.Normal(loc = t[..., :1],
        scale = (1e-3 + tf.math.softplus(0.1 * t[...,1:])),)),])
        model.compile(optimizer='adam', loss=negloglik,metrics=[Model.coeff_dete
        return model

class Probability:
    """
    Class for running a series of simulations at a pre-known
    maximum quantity of error and getting standard deviation
    of predicted parameters from true value.

    Runs numTrials trials for each quantity of error between
    maximum error and zero, then plots and gets coefficient
    of determination.

    The Purpose of this class is to return the standard deviation
    of each parameter based on estiamted gaussian noise in the data
    and try and predict uncertainty range of prediction.

    Parameters
    ----------
    sig : Signal (object), default=False
        Must provide instance of signal class in order to
        build model or use for prediction. Else, will fail
        to train.

    maxError : float, default=5.
        Max standard deviation of gaussian noise added to
        simulations. All other standard devations
        between 0 and max are used to predict.

    numTrials : Int, default=1000
        Number of simulations run for each quantity of standard
        deviation added to simulated response.
```

```
    plotUncertainty: bool, defaulyt=True
        Determine whether or not to plot histrogram and plot
        of predictions with coefficient of determination.

    Attributes
    ----------
    SISO_probability_estimate
        Simulates 1000 system responses with specified
        quantity of noise, predicts response with saved
        model and plots prediction/validation data
        points and coefficient of determination.

    MIMO_probability_estimate
        Simulates 1000 MIMO system responses with specified
        quantity of noise, predicts response with saved
        model and plots prediction/validation data
        points and coefficient of determination.

    """
    def __init__(self,sig=False,maxError=5,numTrials=100,plotUncertainty=True):
        self.plotUncertainty = plotUncertainty
        self.maxError = int(maxError)
        self.deviations = np.arange(0,maxError)
        self.numTrials = numTrials
        self.maxError = sig.stdev
        self.nstep = sig.nstep
        self.timelength = sig.timelength
        self.trainFrac = sig.trainFrac
        self.type = sig.type

        if (self.maxError==5 and numTrials==1000):
            suffix = "Default"
        else:
            suffix = '_error_' + str(self.maxError) + '_nTrials_' + str(numTrial

        self.prefix = self.pd+"/Uncertainty/"
        if self.type=="SISO":
            self.errorCSV = self.prefix+"SISO/Data/propData"+suffix+".csv"
            self.SISO_probability_estimate()
        else:
            self.errorCSV = self.prefix+"MIMO/Data/propData"+suffix+".csv"
            self.MIMO_probability_estimate()

    def SISO_probability_estimate(self):
        """ Simulates 1000 system responses with specified quantity of
        noise, predicts response with saved model and plots prediction
        and validation data  points and coefficient of determination."""

        if not(path.exists(self.errorCSV)):
            print("No simulation for these parameters exists in \
```

```
            Uncertainty data. Proceeding with simulation")

        # Initialize the models that are saved using the parameters declared
        predictor = Model(self.nstep)
        predictor.load_SISO()

        deviations = np.arange(0,self.maxError)

        # Empty arrays to put predictions in
        stdev = np.array([0])
        error=np.array([0])
        b_predicted = np.array([0])
        k_predicted = np.array([0])
        a_predicted = np.array([0])

        # Empty arrays to put true parameters in
        b_true_value = np.array([0])
        k_true_value = np.array([0])
        a_true_value = np.array([0])

        # Produce simulation for each standard deviation below
        # the max standard deviation to add to the gaussian
        # noise after simulation
        for deviation in deviations:
            numTrials = self.numTrials; nstep = self.nstep
            timelength = self.timelength; trainFrac = self.trainFrac

            # then simulates using the initialized model
            sig = Signal(numTrials,nstep,timelength,trainFrac,stdev=deviatio
            sig.SISO_simulation(b_possible_values=[1,10],a_possible_values=[

            # In this case, since we are only loading the model, not trying
            # we can use function simulate and preprocess
            xData,yData = sig.SISO_validation()

            # Function to make predictions based off the simulation
            predictor.predict_SISO(sig,savePredict=False,plotPredict=False)

            # Add predicted values to end of arrays
            error = np.concatenate((predictor.errors,error))
            b_predicted = np.concatenate((predictor.b_model_predictioniction
            k_predicted = np.concatenate((predictor.k_model_predictioniction
            a_predicted = np.concatenate((predictor.a_model_predictioniction

            # Add true values to the end of arrays
            b_true_value = np.concatenate((sig.aVals,b_true_value))
            k_true_value = np.concatenate((sig.kVals,k_true_value))
            a_true_value = np.concatenate((sig.aVals,a_true_value))
            stdev = np.concatenate((np.full_like(predictor.errors,deviation)
```

```
        # Transfer to Pandas before saving to CSV
        sd = pd.DataFrame()
        sd['stdev'] = stdev
        sd['mse'] = error
        sd['b_model_prediction'] = b_predicted
        sd['a_model_prediction'] = a_predicted
        sd['k_model_prediction'] = k_predicted
        sd['b_true_value'] = b_true_value
        sd['a_true_value'] = a_true_value
        sd['k_true_value'] = k_true_value

        sd.to_csv(self.errorCSV, index=False)

    # If the CSV with data already exists, can skip training and plot
    else:
        print("Data exists for the parameters, proceeding to producing uncer
        try:
            sd = pd.read_csv(self.errorCSV).drop(['Unnamed: 0'],axis=1)
            sd.drop(sd.tail(1).index,inplace=True)
        except:
            sd = pd.read_csv(self.errorCSV)
            sd.drop(sd.tail(1).index,inplace=True)

    # Iterate over each of the parameters and plot the histogram
    # as well as the prediction/true parameters
    self.errorDict = {}
    pathPrefix = self.prefix+"SISO/Plots/"
    prefixes = ['b','a','k']
    for (i,prefix) in enumerate(prefixes):
        sd[prefix+'Error'] = (sd[prefix+'Pred']-sd[prefix+'True'])
        h = np.std(sd[prefix+'Error'])
        self.errorDict[prefix] = h

        if self.plotUncertainty:
            plt.figure(dpi=200)
            plt.hist(sd[prefix+'Error'],bins=100,label='Max Error = %i%%' %
            plt.xlabel('Standard Error in '+ prefix)
            plt.ylabel("Frequency Distribution")
            plt.legend()
            savePath = pathPrefix + "histogram_" + prefix + ".png"
            plt.savefig(savePath)

            plt.figure(dpi=200)
            plt.plot(sd[prefix+'True'],sd[prefix+'Pred'],'.',
                     label='Max Error = %i%%' % self.maxError)
            plt.plot(np.linspace(1,10),np.linspace(1,10),'r--',
                     label="r\u00b2 = %.3f" % r2_score(sd[prefix+'True'],sd[
            plt.plot(np.linspace(1,10),np.linspace(1,10)+h,'g--',
                     label="Stdev = %.3f" % h)
            plt.plot(np.linspace(1,10),np.linspace(1,10)-h,'g--')
```

```
            plt.ylabel("Predicted Value of "+prefix)
            plt.xlabel("True Value of "+prefix)
            plt.legend()
            savePath = pathPrefix + "determination_" + prefix + ".png"
            plt.savefig(savePath)

    def MIMO_probability_estimate(self):
        """ Simulates 1000 system responses with specified quantity of
        noise, predicts response with saved model and plots prediction
        and validation data  points and coefficient of determination."""

        if not(path.exists(self.errorCSV)):
            print("No simulation for these parameters exists in Uncertainty data

            # Initialize the models that are saved using the parameters declared
            predictor = Model(self.nstep)
            predictor.load_MIMO()

            deviations = np.arange(0,self.maxError)

            stdev = np.array([0])
            error=np.array([0])
            b_predicted = np.array([0])
            a_predicted = np.array([0])

            b_true_value = np.array([0])
            a_true_value = np.array([0])

            for deviation in deviations:
                numTrials = self.numTrials; nstep = self.nstep
                timelength = self.timelength; trainFrac = self.trainFrac

                # then simulates using the initialized model
                sig = Signal(numTrials,nstep,timelength,trainFrac,stdev=deviatio
                sig.MIMO_simulation(b_possible_values=[1,10],a_possible_values=[

                # In this case, since we are only loading the model, not trying
                # we can use function simulate and preprocess
                xData,yData = sig.MIMO_validation()

                # Function to make predictions based off the simulation
                predictor.predict_MIMO(sig,savePredict=False,plotPredict=False,p

                error = np.concatenate((predictor.errors,error))
                b_predicted = np.concatenate((predictor.b_model_predictioniction
                a_predicted = np.concatenate((predictor.a_model_predictioniction

                b_true_value = np.concatenate((sig.aVals.ravel(),b_true_value))
                a_true_value = np.concatenate((sig.aVals.ravel(),a_true_value))
                stdev = np.concatenate((np.full_like(predictor.errors,deviation)
```

```
        # Transfer to pandas to save to CSV
        sd = pd.DataFrame()
        sd['b_model_prediction'] = b_predicted
        sd['a_model_prediction'] = a_predicted
        sd['b_true_value'] = b_true_value
        sd['a_true_value'] = a_true_value

        sd.to_csv(self.errorCSV, index=False)

    # If the simulation does exist for this set of parameters,
    # load from CSV and plot
    else:
        print("Data exists for the parameters, proceeding to producing uncer
        try:
            sd = pd.read_csv(self.errorCSV).drop(['Unnamed: 0'],axis=1)
            sd.drop(sd.tail(1).index,inplace=True)

        except:
            sd = pd.read_csv(self.errorCSV)
            sd.drop(sd.tail(1).index,inplace=True)

    # Plot histogram and predicted/true parameters
    self.errorDict = {}
    pathPrefix = self.prefix+"MIMO/Plots/"
    prefixes = ['b','a']
    for (i,prefix) in enumerate(prefixes):
        sd[prefix+'Error'] = (sd[prefix+'Pred']-sd[prefix+'True'])
        h = np.std(sd[prefix+'Error'])
        self.errorDict[prefix] = h

        if self.plotUncertainty:
            plt.figure(dpi=200)
            plt.hist(sd[prefix+'Error'],bins=100,label='Max Error = %i%%' %
            plt.xlabel('Standard Error in '+ prefix)
            plt.ylabel("Frequency Distribution")
            plt.legend()
            savePath = pathPrefix + "histogram_" + prefix + ".png"
            plt.savefig(savePath)

            plt.figure(dpi=200)
            plt.plot(sd[prefix+'True'],sd[prefix+'Pred'],'.',
                     label='Max Error = %i%%' % self.maxError)
            plt.plot(np.linspace(1,10),np.linspace(1,10),'r--',
                     label="r\u00b2 = %.3f" % r2_score(sd[prefix+'True'],sd[
            plt.plot(np.linspace(1,10),np.linspace(1,10)+h,'g--',label="Stde
            plt.plot(np.linspace(1,10),np.linspace(1,10)-h,'g--')
            plt.ylabel("Predicted Value of "+prefix)
            plt.xlabel("True Value of "+prefix)
            plt.legend()
```

```
            savePath = pathPrefix + "determination_" + prefix + ".png"
            plt.savefig(savePath)


def get_errors(self):
    """Access standard uncertainty from dictionary,
    return as tuple"""
    return self.errorDict.values()
```