TKP4580 - Chemical Engineering, Specialization Project

Optimization of Heat-Exchanger networks using Gaussian Processs Regression

Written by:

Thomas Edvardsen

thomaedv@stud.ntnu.no

Supervisor: Sigurd Skogestad

Co-Supervisor: Lucas Ferreira Bernardino

Submitted: December 18, 2020

Department of Chemical Engineering

NTNU

Norwegian University of Science and Technology

Abstract

Optimal operation of heat exchanger networks can reduce cost and energy use in the industry. In this project the performance of using Gaussian Processes regression to estimate the optimal valve splits on a heat exchanger network was evaluated. The network consisted of a single input stream that splits to 3 heat exchangers in parallel, with the aim to maximise the temperature of the combined stream out of the heat exchangers by adjust the splits of the stream. Several sets of measurements of a modelled system were created, and optimal valve splits were calculated. The Gaussian process trained on the measurements with the optimal valve splits as output. The results of the testing found that a selection of measurements, called measurement set 2 in this project, provided the best performance with Gaussian Process regression, while also only relying only on temperature measurements of the system. Testing with measurement set 2 gave valve splits that resulted in predicted temperatures within 1.6°C of the real optimal, but almost all the predictions were under 1°C away from the optimal, and over half of them under 0.5°C. The process was trained with 2500 datapoints, but it was shown that 500 training datapoints predictions close to the 2500 samples dataset.

Preface

This project was performed as part of specialization project for students in their first semester of their final year studying for a Masters degree at the Norwegian University of Science and Technology (NTNU). The course was named TKP4580 - Chemical Engineering, Specialization Project. The project was performed through Autumn 2020.

The supervisor was Sigurd Skogestad and co-supervisor was Lucas Ferreira Bernardino. I'm especially grateful to Lucas, as he has given superb guidance and helped provide models and papers to help make this project possible to complete. I also want to thank Sigurd for the feedback provided during our bi-weekly meetings, which has helped out with aiming the projects direction and selecting the implementations that have had the most practical sense.

List of Figures

List of Tables

List of Symbols

Table 0.1: Collection of symbols and their meaning.

Symbol	Meaning	Unit
T	Temperature	[°C]
J	Cost function	[°C]
$w_{\mathrm{h},i}$	Heat capacity of a given stream	[kW/K]
dTlm	Logarithmic middle temperature	[°C]
α, β, γ	Valve splits, gamma depends on the other two	[-]
k_y	Covariance function, also called kernel.	[-]
σ	Noise paramerter, hyper-parameter of RBF kernel.	[-]
ℓ	Lenghtscale, hyper-parameter of RBF kernel.	[-]
Q	Heat transfer	[kW]

Contents

1 Introduction

Optimal operation is critical for both the business and the world. When it comes to heat exchangers, applying them to transfer as much heat as possible can save both money for the operating company as well as reduce energy consumption to make the process ever so slightly greener from an energy perspective. Therefore, good methods to find optimal operating points are important. Over 30% of energy consumption in Norway comes from manufacturing for example. [2]

Machine learning is growing, and heralded as the future in several fields, and process control is no exception to that popularity. [3] Machine learning can train on data that may already exist, and does not require complex modelling and measurements to perform. Ideally one would let the machine learning method figure out system specifics out for itself. If it's data based, then updates can come through new data from the process, without needing to implement changes to a model that governs the optimisation system. One point of note is that if you let something like a neural network train on the data, how do you know it's learned the hidden rules of the process and how well it responds to unexpected outliers? It can be very hard to determine how the neural network determines the output based on just it's weights and network setup. Gaussian processes can help with this, where a co-variance function determines the output based on the training data, as well as gives the covariance back as a measure of certainty in the prediction. With this prediction method, a higher level of trust can be placed on Gaussian Processes. This is also one of the reasons Gaussian Processes are interesting, as they do not suffer from overfitting, and give back interepretable more results. [4]

The aim of the project is to determine if Gaussian Process regression is suitable to be used to control a heat exchanger network. The focus is on a relatively simple setup with one input and one output stream. The input stream is split into 3, where the split ratios are the manipulated variable, and each stream passes through a heat exchangers, before being merged back into the output. The goal is to maximize the temperature of the output stream, using Gaussian Process regression and various measurement taken from the process during modelled operation.

1.1 Scope of work

The main focus of the work is to look at how well the optimal stream splits can be directly predicted by the Gaussian Process. Various measurement sets are tested to see how well it handles the different inputs, focusing on measurements that are easy to make. A closed-loop approach will be investigated, where it will be tested for convergence, given the same operating conditions with updated optimized valve predictions from the Gaussian Process.

2 Theory

2.1 Heat-Exchanger Network

In this setup, there is a unconstrained optimization with the goal of achieving the highest output temperature from a Heat Exchanger (HX) Network. In this report, all temperatures mentioned are in °C.

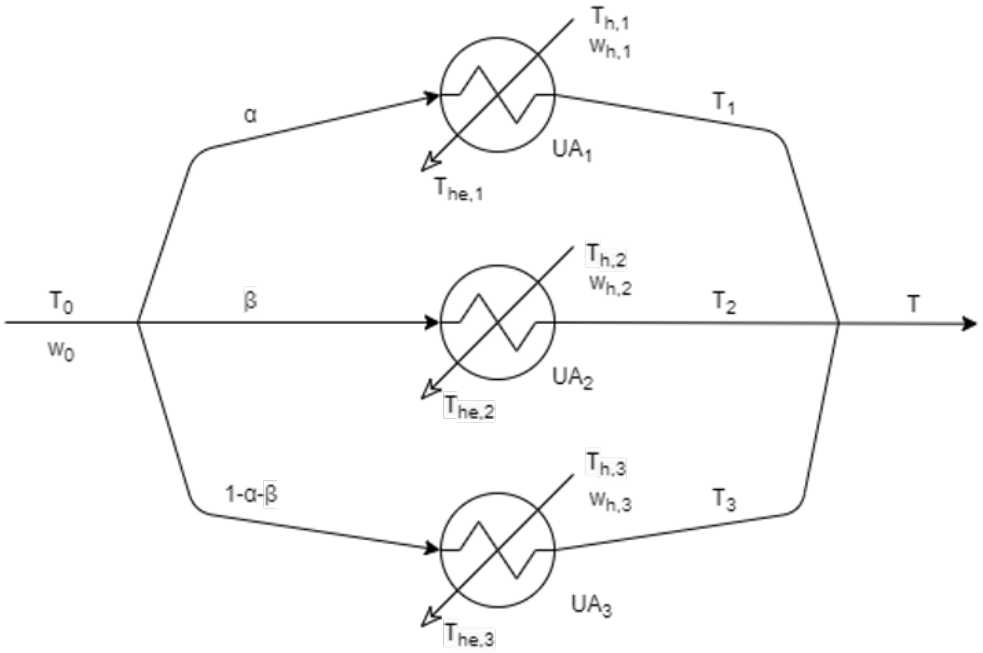

Figure 2.1: Illustration of the Heat Exchanger Network. A input stream is split according to the values α and β, which are the valve splits. Each stream is heated through a Heat Exchanger before merged back into a single stream.

An illustration of the setup is shown in Figure 2.1. The symbols, with the exception of α and β, are all the disturbances of the process which are required to calculate the output temperature using a numerical model, denoted as the real model or the plant in this report. α and β are still required to solve the model, but are considered inputs in this case. More details on that implementation is in Section 3.1. The T_i's describe the temperatures of the respective streams they are attached to, and where T is the output temperature and is considered our cost function that we want to maximize. α and β are the stream splits, where the last stream is merely the remainder of one minus α and β. The UA is the product of the overall heat transfer coefficient and the area of one of the sides of the heat exchanger, and the w is the heat capacity of the stream. The subscript $_i$ is use to denote which of the streams the disturbance applies to. That is, $i \in \{0, 1, 2, 3\}$, where the 0 indexed stream is the cold input stream before being split. The same applies to the subscript $_{\mathrm{h},i}$ where the h denotes that it's for the hot stream going into the HX, and he for the hot stream going out of that HX.

Some of the key model equations are:

$$T = T_1 \cdot \alpha + T_2 \cdot \beta + T_3 \cdot \gamma \tag{2.1}$$

$$\gamma = 1 - \alpha - \beta \tag{2.2}$$

$$1 = \alpha + \beta + \gamma \tag{2.3}$$

Where in Equation 2.2 the γ is the remaining valve opening, however since it's determined by the other two, it's not considered worth including outside the numerical model implementation.

$$dTlm1 = \left((T_{h,1} - T_1^*) \cdot (T_{he,1} - T_0) \cdot \frac{1}{2}((T_{h,1} - T_1^*) + (T_{he,1} - T_0)) \right)^{\frac{1}{3}} \tag{2.4}$$

$$dTlm2 = \left((T_{h,2} - T_2^*) \cdot (T_{he,2} - T_0) \cdot \frac{1}{2}((T_{h,2} - T_2^*) + (T_{he,2} - T_0)) \right)^{\frac{1}{3}} \tag{2.5}$$

$$dTlm3 = \left((T_{h,3} - T_3^*) \cdot (T_{he,3} - T_0) \cdot \frac{1}{2}((T_{h,3} - T_3^*) + (T_{he,3} - T_0)) \right)^{\frac{1}{3}} \tag{2.6}$$

Equations 2.4 to 2.6 define the logarithmic middle temperatures for each HX. And the following equations describe the heat transfer equations. Where Ts is the environment temperature and the h_i is the heat loss coefficients for each stream.

$$Q_1 = w_0 \cdot \alpha \cdot (T_1^* - T_0) \tag{2.7}$$

$$Q_2 = w_0 \cdot \beta \cdot (T_2^* - T_0) \tag{2.8}$$

$$Q_3 = w_0 \cdot \gamma \cdot (T_3^* - T_0) \tag{2.9}$$

$$Q_1 = w_{h,1} \cdot \alpha \cdot (T_{h,1} - T_{he,1}) \tag{2.10}$$

$$Q_2 = w_{h,2} \cdot \beta \cdot (T_{h,2} - T_{he,2}) \tag{2.11}$$

$$Q_3 = w_{h,3} \cdot \gamma \cdot (T_{h,3} - T_{he,3}) \tag{2.12}$$

$$Q_1 = UA_1 \cdot dTLm1 \tag{2.13}$$

$$Q_2 = UA_2 \cdot dTLm2 \tag{2.14}$$

$$Q_3 = UA_3 \cdot dTLm3 \tag{2.15}$$

$$Qloss_1 = w_0 \cdot \alpha \cdot (T_1 - T_1^*) \tag{2.16}$$

$$Qloss_2 = w_0 \cdot \beta \cdot (T_2 - T_2^*) \tag{2.17}$$

$$Qloss_3 = w_0 \cdot \gamma \cdot (T_3 - T_3^*) \tag{2.18}$$

$$Qloss_1 = h_1 \cdot \alpha \cdot (Ts - T_1) \tag{2.19}$$

$$Qloss_2 = h_2 \cdot \beta \cdot (Ts - T_2) \tag{2.20}$$

$$Qloss_3 = h_3 \cdot \gamma \cdot (Ts - T_3) \tag{2.21}$$

It can must noted that heat loss is neglected in this study. Thus $Qloss_1$, $Qloss_2$, and $Qloss_3$ are reduced to zero through setting Ts, h_1, $h_2$and h_3 to zero. Note that T_i^* are the temperatures out of each HX, before heat loss is applied afterwards, which leads to the new temperatures T_i. This means that T_i^* and T_i are equal when the heat loss is zero, as in this case.

2.2 Surrogate optimization

Surrogate modelling is the way of optimization, by trying to quickly find local or global optima for operation. This is done through random or controlled sampling of the design space, which in our case would be for the ranges of expected disturbances and valve openings. Through the use of surrogate modelling, a surrogate or approximate model is made to predict the optimal faster or more easily compared to an accurate model of the process that we aim to optimize. They bring the advantage of getting close to the real plant, but not having to deal with modelling or working with disturbances that are hard to measure. Such as in the case of the heat exchangers, where the temperatures are much easier to measure than the heat capacities, or the universal heat transfer coefficients. As long as the approximate model performs well enough, this can remove the need for complicated measurements and may speed up the optimization implementation.

Compared to data driven methods, a standard model based approach can give much more accurate predictions, even if sometimes costly. The problem is that they require good knowledge of the process and usually a wide array of measurements to sufficiently know the state of the system and predict the optimal. As mentioned, disturbances can be hard to measure, and simplifying the model to not rely on those disturbances may not give the accuracy that is desired. There is also the potential for numeral issues with models, when solving complicated system to predict optimal operation, one might run into the risk of non-convergence for particularly hard systems.

There is the option of using a good accurate model to generate data and then train the approximate model an that. By picking easily measurable information in a real plant, and training the approximate model on that, one could sidestep the problems of a model based approach if the resulting approximate model performs. Since information can be generated beforehand and trained on, time is not a problem. Then, it's just the need to put that approximate model into practice and measure the performance in the real plant. If getting measurements from a model is not feasible, real measurements can be used to train an approximate model as well.

2.3 Machine Learning

Machine learning is the method of getting machines to improve through experience.[5] Machine learning is applied everywhere in modern times, from telling which emails that are spam, finding out what advertisements are most fitting to you, and even learning to drive. Models are created and trained on the experience they get, which is usually sampled data, which for spam detection would be emails labeled as legitimate or spam.

Detecting emails as spam is a classification problem, where it outputs a discrete answer. The other type of problem is regression, where outputs are not discrete, for example a machine learning model that takes temperature data and predicts the temperature for tomorrow. The goal of machine learning is that it learns the nature of the "process", the hidden rules, to make good predictions, not only on the information it's trained on, but also new data.

It is therefore important to have separate training and test/validation datasets. Because you may run into the risk of the model learning the specifics of what you are training it on, not the underlying rules, leading to predictions that are wrong for any new data you test the model with. The test or validation dataset lets one confirm that the trained model works on information it has not "seen" before. It stands to reason then that machine learning can also be applied to optimization, letting it learn the nature of the process and "getting a hang off" what changes give an optimal output.

One can also divide machine learning into two categories, parametric and non-parametric. Many think of machine learning as the Artificial Neural Nets (ANNs), which can have hidden layers and nodes, which aim to simulate decisions similar to neurons in the human brain. ANNs are parametric, which means they have weights on these nodes, which influence the prediction. The weights are adjusted through the training process. Non-parametric machine learning, such as Gaussian Processes, do not learn weights, but work off of the training data and some hyper-parameters. For the case of ANNs, they hyper-parameters are the selected network layouts, the method of estimating loss (see section 2.4.2), and how it's set to optimize the weights.

2.4 Gaussian Processes

For the case of Gaussian Processes (GP), the approach is supervised learning, where a input-output mappings are established from empirical data. GP uses a form of lazy-learning where the learning from the training data is done when a test input is given to make a prediction. This is different from ANNs which training their weights and only rely on the weights and layout of the network, GP requires the training data or a optimized selection of it, to make test predictions.

The general notation is that x denotes the input, and y denotes output or target from a machine learning model. Both x and y can be vectors. A dataset of is thus composed of the following "observations", $\mathcal{D} = \{(x_i, y_i)|i = 1, ..., n\}$, where n is the number of samples. Now, given the dataset, $\mathcal{D}$, how does one go from x to y. The approach is to move from $\mathcal{D}$ to a function f which makes predictions for all possible inputs. This require some assumptions on characteristics of the underlying function (our actual model or optimization case) to work. One way to do so is to give a prior probability to every possible function, where higher probabilities are given to functions assumed to be more likely to fit the problem. However this isn't easy to do as there can be infinite sets of possible functions to use. The Gaussian process is what deals with this issue. GP makes use of a generalization of the gaussian probability distribution. Simply put, a function can be considered as an infinitely long vector that defines the solution $f(x)$ for a given x.[1]

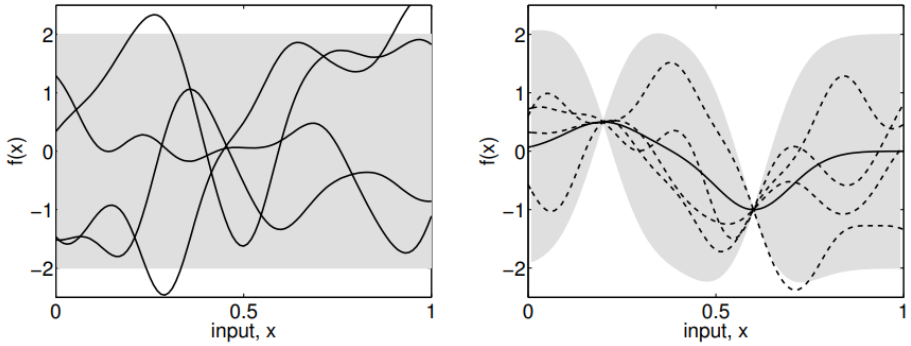

Figure 2.2: The prior distribution show some random functions drawn from it, while the posterior shows after two datapoints from a dataset $\mathcal{D}$ have been introduced. The thick line being the mean of the dotted ones, and the shaded area twice the standard deviation for each input value. [1]

For a 1-D regression problem, given a set of sample functions randomly picked from the prior distribution (Figure 2.2 (a)) and a dataset with points, we only want to consider function which pass through those datapoints (or close to them). Using this we can find the posterior over the functions, as seen in Figure 2.2 (b). Take note how variance decreases close to the datapoints. Adding more datapoints would adjust the mean to align with those datapoints as well, as well as decrease the variance around those. Through this we can find predictions and get the mean and variance back. [1] To be more precise, the goal is to predict the expectation $\mathbb{E}\left[y(x_*)|x_*, \mathcal{D}\right]$ and the variance $\text{cov}\left[y(x_*)|x_*, \mathcal{D}\right]$ for a test input x_*.

2.4.1 Kernel

At the core of the GP is the coveriance functions that describe the correlation between the datapoints, and thus the choice directly affects the nature of the data you have. The coveriance function is called the kernel. Within GPy, the python framework used, you can have the kernel be a sum of coveriance functions as well, to describe more complex relations. However, for this project, only the RBF kernel was used. The RBF kernel is also known as the squared exponential and is shown in Equation 2.22.

$$k_y(x_p, x_q) = \sigma_f^2 \exp\big(-\frac{1}{2 * \ell^2}(x_p - x_q)^2\big) + \sigma_n^2 \delta_{pq} \tag{2.22}$$

Where the kernel in this case is referred to as k_y, with x_p and x_q are datapoints and δ_{pq} is the Kronecker delta, which is equal to 1 if p = q and 0 otherwise. The remaining variables σ_f^2, σ_n^2 and ℓ are hyperparameters for the RBF kernel. They are described as the signal noise, input noise and the lenghtscale. Varying these parameters affects the prediction. However, the optimization of these parameters have been left to the GP framework. Hyperparameters are important parts of the kernel, manually picking the wrong lenghtscale would cause it to incorrectly take data far away into account, or ignore data it should not.

2.4.2 Cost and Loss

As was mentioned, the goal is to maximize the temperature T out of the HX network, which we can put on the form:

$$J = T \tag{2.23}$$

where J is the cost.

Loss is usually the metric which measures the performance of machine learning models, and in traditional ANNs this loss is used to update the weights that decide the output. The loss would be the on objective one would try to minimize or maximize through the training process. However in the case of GP, which is non-parametric and without weights, the loss is just a measure of the error of the cost function. The general loss is simply defined as the difference between the predicted and optimal cost.

$$\text{Loss} = J^* - J \tag{2.24}$$

Where J^* is the optimal cost found from the accurate model. Traditionally Mean Squared Error (MSE) has used for loss, but given the low residuals from the optimization, the Root Mean Square Error (RMSE) is used in this project when comparing the cost of all the samples for a prediction set.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(J_i^* - J_i)^2} \tag{2.25}$$

Where n is the number of samples in the set of predictions.

3 Implementation

3.1 The "Real" Model

A python implementation of the heat exchanger network was made using the CasADi package. This was used to generate the "real plant" information used to train and test the Gaussian Process implementation. Multiple datasets were made with different sample counts. The test dataset was made with higher disturbance variations to test the Gaussian process ability to extrapolate beyond the disturbances of the training dataset. The code for the real model is shown in appendix B.4, and the script to generate measurement sets is given in appendix B.3.

3.2 The Gaussian Implementation

The implementation was created in Python using GPy [6], a framework designed to perform Gaussian Process machine learning algorithms. The core flow for the GP regression is detailed as follows, and the bulk of the code is within the `g_process.py` file in appendix B.1.

(i) The test and training data is loaded into the GPModel instance.

(ii) The training input is normalized to between 0 and 1, and the training output can be normalizedd if selected.

(iii) A RBF kernel is initialized based on the shape of the training input, and an Intrinsic Coregionalization Model (ICM) kernel is set up with the RBF kernel as argument. This due to the framework requiring this setup for multiple output predictions.

(iv) The model for multioutput regression is set up from the training data and the ICM kernel.

(v) The optimization of the model is performed to the best seen solution from the frameworks point of view. This is run on the training dataset internally, and optimizes the hyperparameters mentioned in Section 2.4.1.

The following steps can be repeated with new test data, as is the case for the closed loop implementation.

(vi) Test input is fed to the GPModel instance and normalized.

(vii) A test prediction is made by feeding the model the test data.

(viii) The test prediction is reverse normalized if the training output was normalized, and then returned.

A simple framework was built around this to facilitate testing multiple datasets and options such as output normalisation and adding noise. It is also what handles some of the processing such as calculating the model measurements if a measurement set is used for regression. In a baseline approach, disturbances are

used as input for regression. The simple framework also handles cost and error calculations, closed-loop implementation, and plotting of various results.

Due to some implementation issues the GP regression is performed in a multiprocess instance in Python, as the GPy framework would not execute properly when multiple models were created.

3.2.1 Optimal Valve splits

The main focus was on the prediction of the optimal valve splits, and the closed-loop implantation was adopted to this approach. This section details some of the specifics of the implementation.

The datasets contained all the disturbances given for the process (Figure 2.1), as well as a pair of random valve openings and the optimal valve openings. This allows for consistency between when calculating measurement sets, as the random valve openings can be used to measure the plant at a non-optimal state to train the GP model with. There is also implemented a weighting system, such that portions of the random valve openings can be adjusted closer to the optimal to verify if they affect the performance. The implementation is shown in the code in appendix B.2.

3.2.2 Closed loop

The closed loop approach is that once a prediction of the optimal valve splits had been done, the predictions were used to calculate new measurements for those openings, and then fed back into the GP model to give a new prediction. This is repeated until the change in the predicted values is below a threshold. Once we have reached the threshold, we consider it converged. Convergence is important given that we would be relying on it to bring the HX network towards the optimum operating point when disturbances don't change, and change it to a new optimum when disturbances change.

For this project the tolerance condition was calculated as the max absolute change in the predicted outputs since the last iteration. The tolerance was set to 10^-8 for all closed loop iterations. Iterations start counting from zero, and are stopped at the 20th iteration if the tolerance has not been met, due to long a runtime of the closed loop calculations.

3.2.3 Noise cases

There was tests for how noise affected prediction, and three cases were tested. The noise applied was gaussian with a range of ± 1 °C.

Case 1 is that there is no measurement error.

Case 2 is with noise on the test data, signifying the real plants noisy measurements while the training samples were noise free based on the assumption that training data came from a model prediction without noise on the measurements.

Case 3 is where both the training and testing data has introduced noise, a scenario where the training data either has simulated noise or where noisy training measurements come from a real process.

3.2.4 Measurement sets

Several measurement sets were created, either taken from disturbances, the inputs or the state of the model. They can be seen below, where n is number of samples for the dataset, and in this case, i is the index of the sample in the dataset. The datasets are setups as pairs of sets, where the first of the sets is the inputs, and the second is the output. The star in α_i^* and β_i^* denotes they are the ideal valve splits, which sets them apart from the α_i and β_i which are merely the valve opening at the time of "measurement". Ideally, with the closed loop approach, α_i and β_i would converge to α_i^* and β_i^*.

$$
\begin{aligned}
n &= \{500, 2500\} \\
\mathcal{D}_{MS1} &= \{(\{T_{0,i}, T_{1,i}, T_{2,i}, T_{3,i}, T_{h1,i}, T_{h2,i}, T_{h3,i}\}, \{\alpha_i^*, \beta_i^*\}) \| i = 1, ..., n\} \\
\mathcal{D}_{MS2} &= \{(\{T_{0,i}, T_{h1,i}, T_{h2,i}, T_{h3,i}, T_{he1,i}, T_{he2,i}, T_{he3,i}\}, \{\alpha_i^*, \beta_i^*\}) \| i = 1, ..., n\} \\
\mathcal{D}_{MS3} &= \{(\{T_{0,i}, T, T_{he1,i}, T_{he2,i}, T_{he3,i}, w_0, wh_1, wh_2, wh_3\}, \{\alpha_i^*, \beta_i^*\}) \| i = 1, ..., n\} \\
\mathcal{D}_{MS4} &= \{(\{T_{0,i}, T, T_{h1,i}, T_{h2,i}, T_{h3,i}, \alpha_i, \beta_i\}, \{\alpha_i^*, \beta_i^*\}) \| i = 1, ..., n\}
\end{aligned}
$$

Optimization that run close to the optimal should be possible through ordinary optimizations methods such as controlling a cost gradient, with just temperature measurements.[7] Along with that, temperatures are easy to measure and would save a lot effort on the measurement side of implementing a control system. Thus measurement set 1 and 2 (MS1 and MS2) are purely temperature based. On the other hand, from a regression point of view the correlations between the measurements and the prediction may be worse, so in measurement set 3 (MS3), the heat capacity of the hot streams are included as part of the measurements. Finally, one can on the assumption that telling the system the current position, both in terms of what the controlled variable currently is, and where in terms of "regression space", would allow the GP model to more easily aim for the optimal prediction values. Thus the valve openings are included in measurements set 4. (MS4)

There was generated two training datasets with 500 and 2500 samples, and one shared test dataset of 2500 samples. The test dataset contains disturbances that were up to 20% larger than those in the training dataset, to test if the GP model had sufficient capability to extrapolate beyond the training data. During runtime, the random valve openings were used to generate the measurement sets from the real plant. From then on, in the closed loop approach, the predictions were used to generate the next set of measurements until convergence. It's also during the measurement generation that noise is added, if enabled.

As mentioned in the section 3.2, there was implemented a method of shifting the random valve splits closer to optimal values. The reasoning being that given most operation being close to the optimal, more points

would be needed there to have accurate predictions, especially in the case for the close loop where we want accurate convergence. This metric was tuned empirically to get the most reasonable results, and focus as many points as possible closer to the optimum, while still retaining convergence.

3.2.5 Normalization

It was looked into if normalizing the output, and applying a reverse transform on the test predictions would increase the accuracy of the predictions. Normalization of the output was not applied in any other test cases. The inputs to the GP models were always normalized. Normalization in all cases were to values between 0 and 1.

4 Results

A selection of key results are provided here, while full iteration results is provided in A. Datasets are provided in their own files, along with code which is also in appendix B. It is noted that the number of training samples are shown in the plots for convenience, while the test samples are always kept at 2500.

4.1 Optimal valve prediction

4.1.1 Baseline

To demonstrate that the model works, a baseline noise free case had to be created, where the training data was all of the disturbances of the model, and the output was the predicted optimal valve splits. This was generated using only 500 training samples as it proved more than sufficient to demonstrate the GP performance.

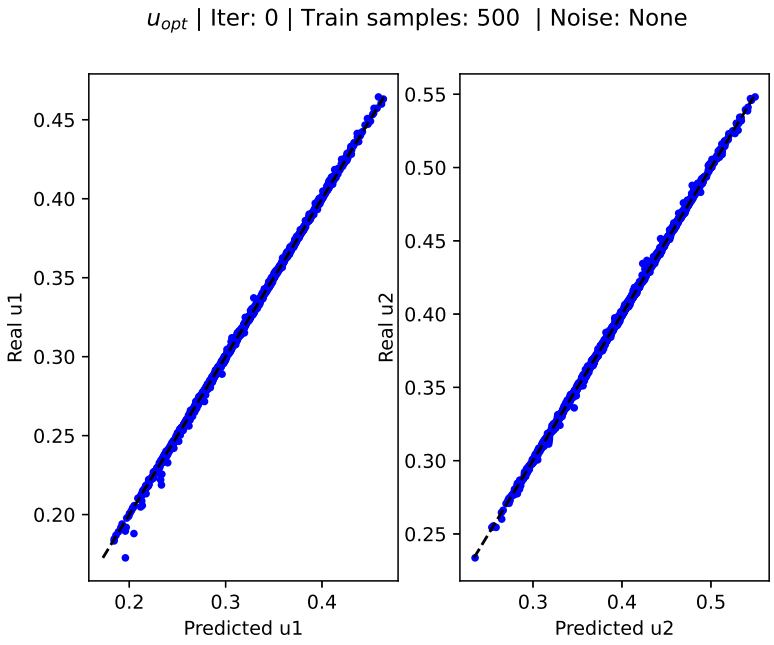

Figure 4.1: Baseline: The plot of the real values over the predicted values. Perfect performance would be everything aligned on along the dotted diagonal.

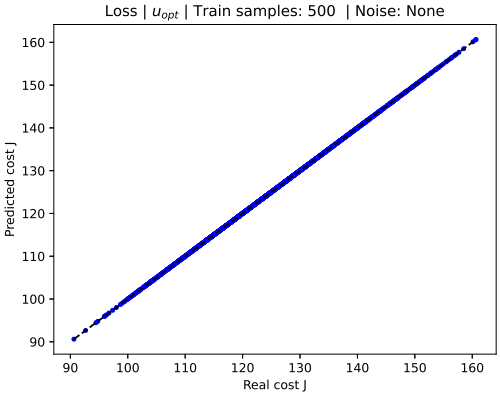

Figure 4.2: Baseline: The plot of the loss of the cost. Since the optimal temperature is the highest, everything should as close to or below the dotted line shown.

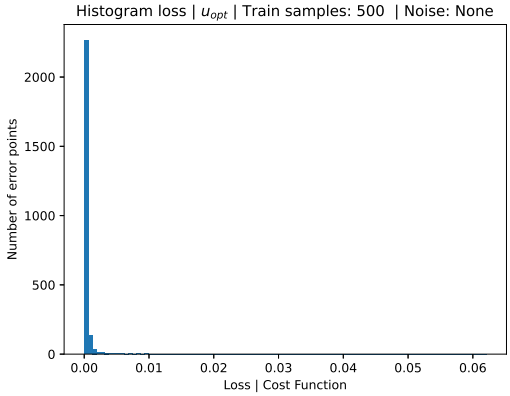

Figure 4.3: Baseline: Histogram of difference between the real and predicted. It can be seen most of the errors are very close to zero.

As can be seen in Figures 4.1 and 4.2, almost all of the 2500 test datapoints are closely aligned with the diagonals. There are a few small outliers at the extremes, but based on the histogram in 4.3 almost all the predictions are extremely close to the optimal. Given the measurements it can be argued that this approach would perform within well expectations. However, at that point one might as well use the model to numerically solve the system and get the optimal configuration that way, with the added benefit of easily updating the model if major changes happen to the process, unlike the GP approach which would also need to have new data for training.

Table 4.1: Baseline prediction using all disturbances, with different noise cases. Trained on 500 samples.

Loss (RMSE)		
1. No Noise	2. Test Noise	3. Train Noise
0.001 88	0.016 55	0.022 96

The RMSE for the baseline run is detailed in Table 4.1. Runs with some added noise also showed some degraded performance. However, looking at these cases in more detail not too useful as they are unrealistic as real world approaches, so they have not been evaluated any further. The RMSE scores for noise cases were included as to be relative comparison from ideal to real world predictions from measurement sets in the sections below.

4.1.2 Normalization on output

First off, the RMSE was calculated for two closed-loop runs on measurement set 2, one with output regularization and another without it.

Table 4.2: Closed loop loss (RMSE) prediction for MS2. Measurement errors were enabled on both training and test data. Did not converge in 20 iterations.

Iteration	Normalized Y	Not Normalized Y
0	0.292 213 92	0.291 159 11
1	0.207 649 01	0.206 694 15
2	0.224 236 56	0.223 521 44
3	0.218 748 50	0.217 848 72
16	0.220 025 26	0.219 182 00
17	0.220 025 16	0.219 181 83
18	0.220 025 22	0.219 181 93
19	0.220 025 18	0.219 181 87

From Table 4.2, the iterations show that normalizing input slightly reduces the prediction accuracy, and it's thus not advised. Since the output are between 0 and 1 to begin with, normalizing it to the same range was not expected to provide any significant improvement.

4.1.3 Noise cases

Since both 500 and 2500 samples were tested, the largest dataset was used for the main discussion and noise comparisons. Extended results are shown in Appendix A, while comparisons of the different dataset sizes are given in section 4.1.4. Along with the tables provided, graphs like the baseline are provided, with special focus on results that stood out.

Table 4.3: Compact table of loss for each noise case for measurement set 1. Trained on 2500 samples. Last iterations shown is where convergence was reached. Measurements for all iterations are shown in Table A.1

MS 1	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.456 699 40	0.458 425 95	0.457 342 56
1	0.465 558 24	0.467 608 42	0.465 708 90
2	0.466 581 35	0.468 651 27	0.466 634 73
6	0.466 646 82	0.468 718 27	0.466 692 36
7	0.466 646 82	0.468 718 27	0.466 692 36

For measurement set 1, convergence is observed after just 8 iterations. Case 2, with only test noise has been picked for visual comparison, but all behave the same, with just slight changes in the RMSE. There is an observed and expected trend of lowest error for the noise free case, and worst for the test noise only case.

We can see that applying the noise on training improved performance. The errors however are relatively the same for all the cases.

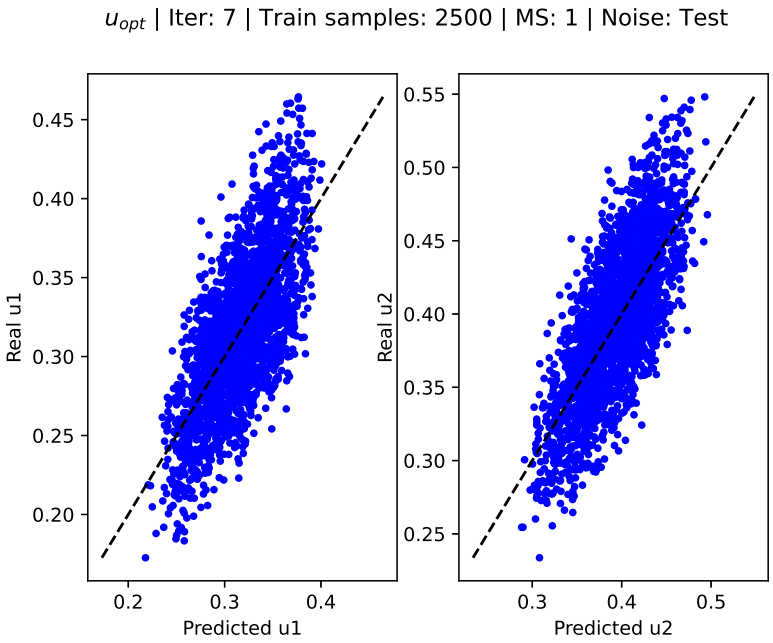

Figure 4.4: MS1: plot of the real values over the predicted values. 2500 samples used for training.

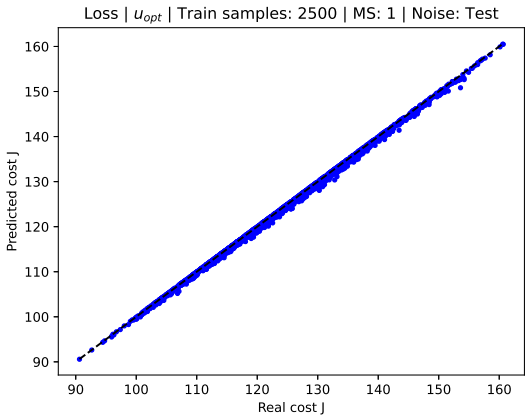

Figure 4.5: MS1: plot of loss of the cost. Since the optimal temperature is the highest, everything should as close to the diagonal.

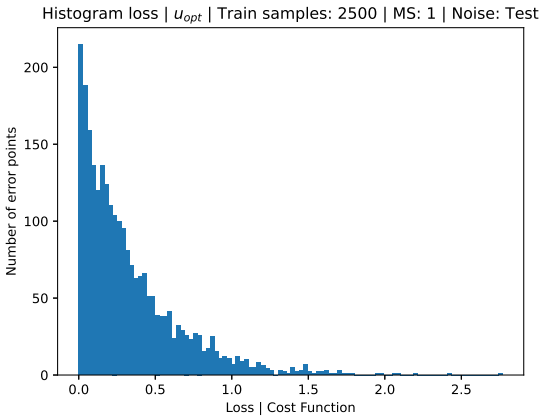

Figure 4.6: MS1: Histogram of difference between the real and predicted loss.

As can be observed for the loss distribution in Figure 4.5, the majority of the points are located close to the optimal. The errors seen in Figure 4.6 show that most of the datapoints are below 1°C, but some outliers stretching as far to as almost 3°C below the optimum point of operations. Since this is without training noise, similar but slightly better predictions would apply to Case 1 and 3.

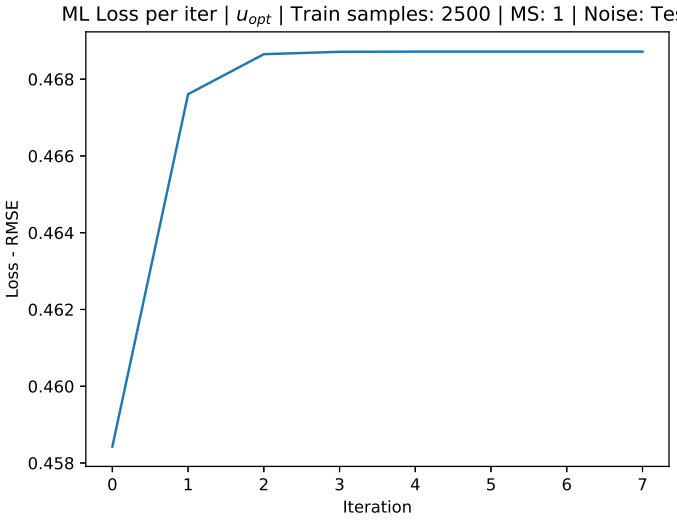

Figure 4.7: MS1: Plot of RMSE loss over each iteration.

Finally, a plot is included to show how the predictions converge in the closed loop approach. In Figure 4.7 we see that the convergence is consistent, and relatively fast. However the point of convergence is higher up than the initial prediction. This is discussed in more detail in section 5.1, but the main assumption is that near the optimal the sample density is not sufficient to make a more accurate prediction. The trend in convergence will be looked at for the other measurement sets as well.

Table 4.4: Compact table of loss for each noise case for measurement set 2. Trained on 2500 samples. Iteration stopped after not reaching the tolerance for convergence at 20 interactions. Measurements for all iterations are shown in Table A.2

MS 2	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.284 660 44	0.290 504 89	0.291 154 63
1	0.201 940 21	0.205 108 59	0.206 691 28
2	0.219 144 98	0.222 869 11	0.223 518 50
17	0.214 434 29	0.218 080 54	0.219 178 78
18	0.214 434 66	0.218 080 74	0.219 178 88
19	0.214 434 39	0.218 080 62	0.219 178 82

For measurement set 2, none of the cases converged properly, the change in the last iteration is still relatively low around 10^{-7}. For this set of results, case 3 with measurement noise applied to both the training and testing has been used for visual representation. The differences between the cases however, are again visually non-important.

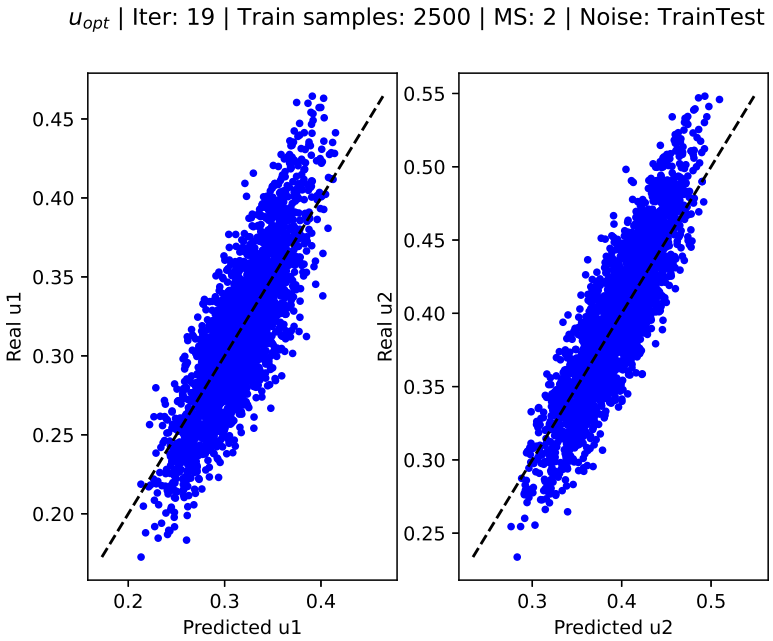

Figure 4.8: MS2: plot of the real values over the predicted values. 2500 samples used for training.

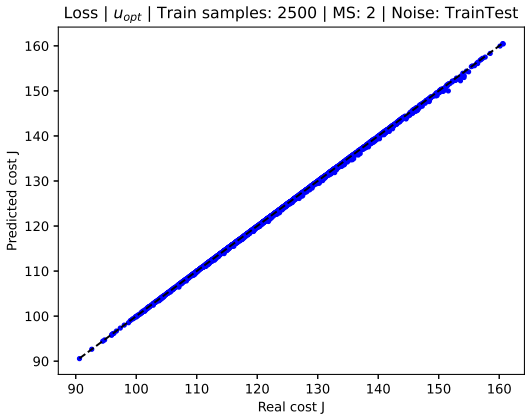

Figure 4.9: MS2: plot of loss of the cost. Since the optimal temperature is the highest, everything should as close to the diagonal.

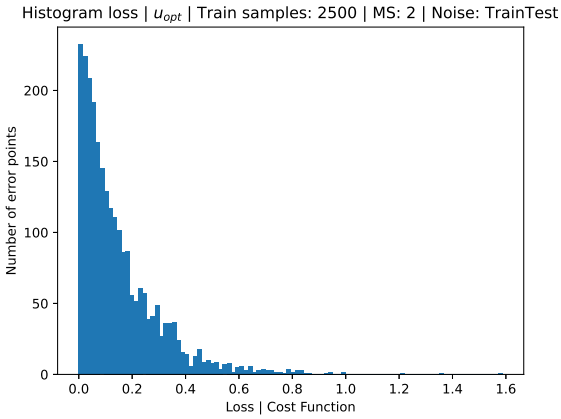

Figure 4.10: MS2: Histogram of difference between the real and predicted.

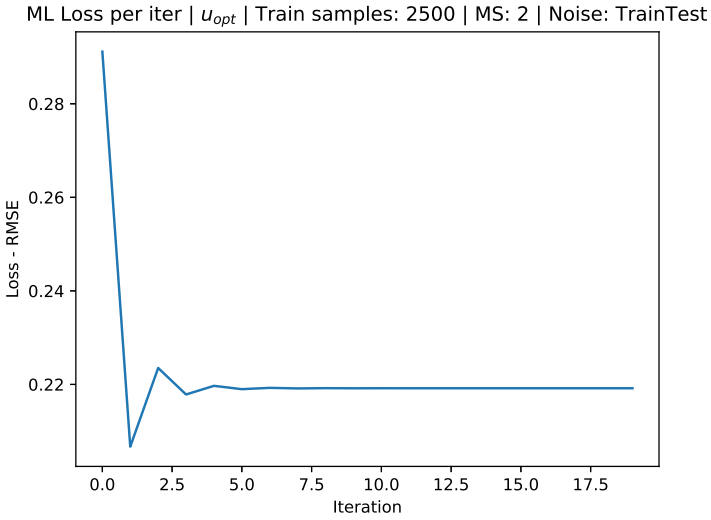

Figure 4.11: MS2: Plot of RMSE loss over each iteration.

Overall, MS2 is the best performing measurement set. The RMSE is the lowest and from the histogram in Figure 4.10, we can see that almost all the datapoints are within 1°C of the optimum, with most below 0.5°C. The interesting thing however is how it converges in Figure 4.11, with the second iteration having the highest accuracy. After that it jumps around and stabilizes around a slightly higher point. This is speculated to be similar to MS1, except that it now "circles" the optimal where the prediction shots into a new area, gradually approaching a steady solution. This could be caused by some points moving further from the optimal after getting very close to it as in iteration 1. Based on Figure 4.9, the predictions are very close to the optimal, and looks a fair bit denser than Figure 4.5 from MS1, as expected with the lower RMSE.

Table 4.5: Compact table of loss for each noise case for measurement set 3. Trained on 2500 samples. Missing values means convergence was detected earlier. Iteration stopped after not reaching the tolerance for convergence after 20 iterations, for the case with only test noise applied. Measurements for all iterations are shown in Table A.3

MS 3	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.290 814 15	0.302 511 04	0.299 119 99
1	0.220 568 12	0.229 063 49	0.227 277 26
2	0.230 636 01	0.239 809 92	0.237 484 31
16	0.228 963 76	0.237 953 16	0.235 784 59
17	-	0.237 953 16	0.235 784 59
18	-	0.237 953 16	-
19	-	0.237 953 16	-

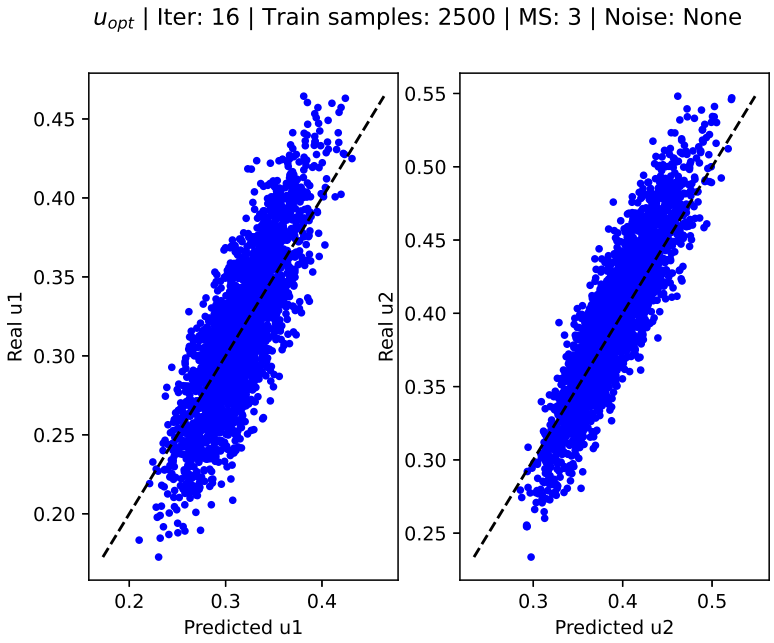

Figure 4.12: MS3: plot of the real values over the predicted values. 2500 samples used for training.

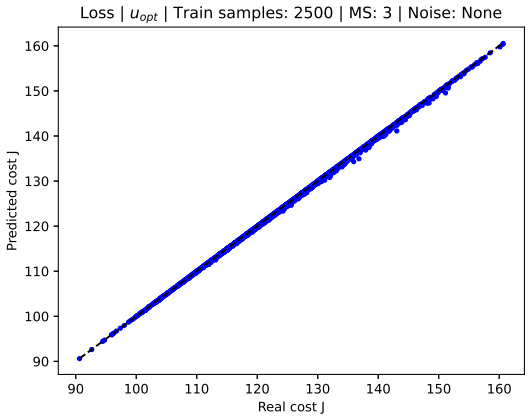

Figure 4.13: MS3: plot of loss of the cost. Since the optimal temperature is the highest, everything should as close to the diagonal.

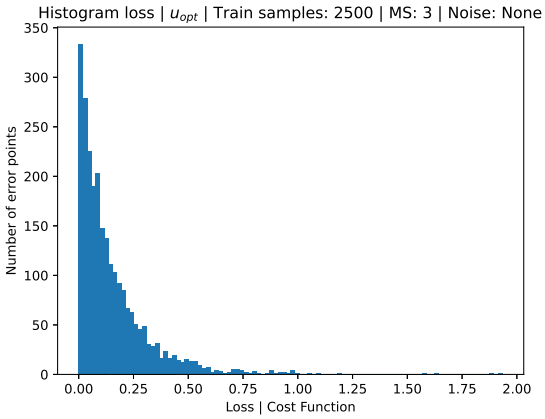

Figure 4.14: MS3: Histogram of difference between the real and predicted.

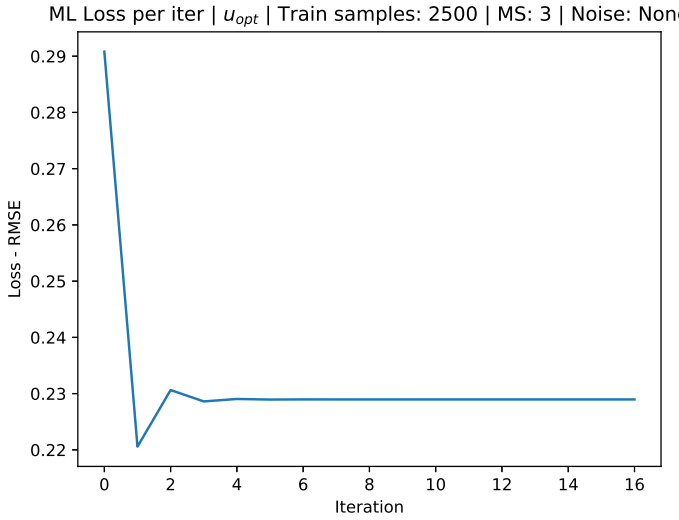

Figure 4.15: MS3: Plot of RMSE loss over each iteration.

Apart from the test case, MS3 converged at iteration 16, while the one that did not had a highly stable RMSE at the end which can be seen in Table 4.5, which probably means there is few points that didn't stabilize before the iteration limit. From Figure 4.12 we see that there was no points which drastically diverged away from the rest however. MS3 converged with a RMSE not too far from measurement set 2, and did much better than MS1. Results are very similar to MS2, with the predicted cost almost entirely under 1°C from the optimal, and most of it being below 0.5°C as seen in Figure 4.14.

Table 4.6: Compact table of loss for each noise case for measurement set 4. Trained on 2500 samples. Iteration stopped after not reaching the tolerance for convergence at 20 iterations. Measurements for all iterations are shown in Table A.4

MS 4	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.218 891 75	0.220 646 41	0.219 080 34
1	0.370 358 80	0.373 660 98	0.374 139 84
2	0.446 822 09	0.451 062 44	0.455 208 88
17	0.511 000 80	0.516 383 62	0.529 146 53
18	0.511 001 62	0.516 384 52	0.529 149 49
19	0.511 002 05	0.516 384 99	0.529 151 31

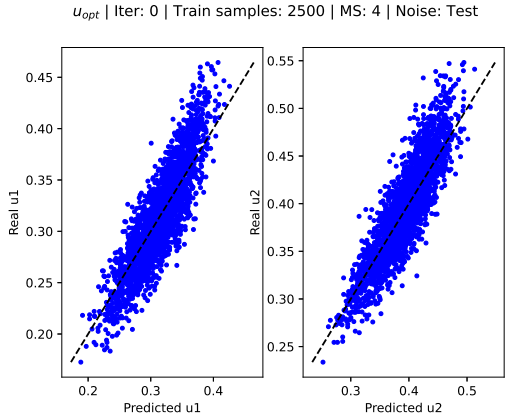

Figure 4.16: MS4: plot of the real values over the predicted values. Iteration 0. 2500 samples used for training.

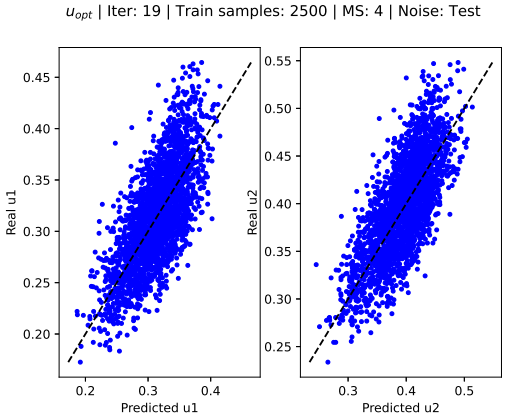

Figure 4.17: MS4: plot of the real values over the predicted values. Iteration 19.2500 samples used for training.

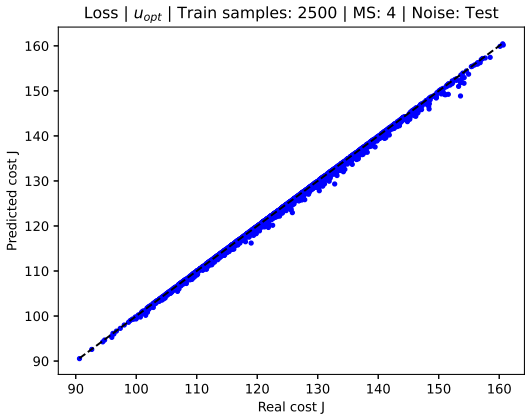

Figure 4.18: MS4: plot of loss of the cost. Since the optimal temperature is the highest, everything should as close to the diagonal.

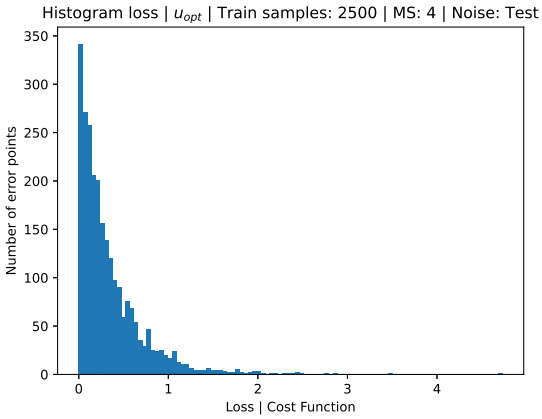

Figure 4.19: MS4: Histogram of difference between the real and predicted.

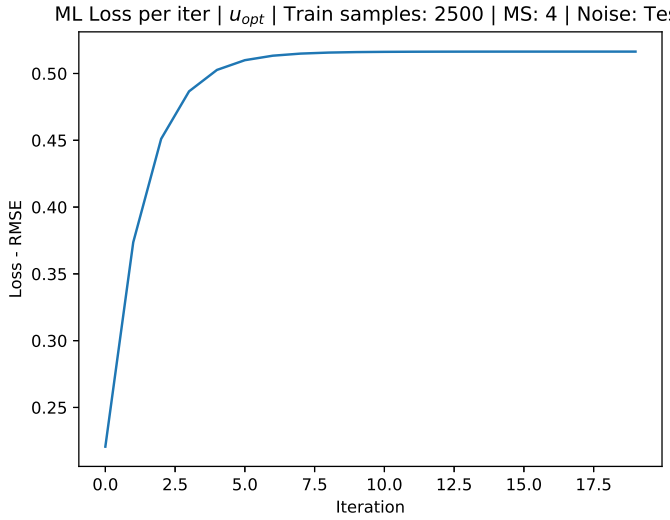

Figure 4.20: MS4: Plot of RMSE loss over each iteration.

Measurement set 4 is dataset that shows the most interesting response, as it converges significantly worse than all the other sets. The first iterations outperforms the following ones, and is close to best results of MS2 and MS3. However the performance worsens rapidly and in the last iteration as seen in Table 4.6, the RMSE increases a little every iteration. In terms of running process optimization, this could lead to the process converging quickly before slowly drifting away from the best state and then eventually making a jump towards back to the optimal before repeating. The stronger divergence is assumed to be caused by the inclusion of the valve opening in the prediction. While the focus on more datapoints close to the optimal, it either was not enough or it was perhaps too focused. A big issues is that MS4 diverged if not enough datapoints were located far away from the optimal. Thus the divergence could be caused by lack of datapoints to properly control for the valve openings when the plant is operating with various large disturbances and very non-optimal valve openings.

In the end though, the convergence ended up at worse performance than the other measurement sets, with some good amount of points stretching out beyond 1°C off the optimal, as well as some extremes at above 4°C off. While the one-shot performance is impressive, it seems dangerous to try to apply for optimization given that most of the operations is at the optimal, and it's not desirable to have a convergence around an optima that is worse than the ones from the other measurement sets.

4.1.4 Number of datapoints

Using the two training datasets consisting of 500 and 2500 datapoints respectively, the change in performance were tested to make an attempt to establish how much data it would be reasonable to collect, or at the very least establish an upper limit.

Table 4.7: Compact table of loss for each noise case for measurement set 1. Trained on 500 samples. Missing values means convergence was detected earlier. Full table in Appendix A.5

MS 1	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.466 118 29	0.467 188 02	0.467 142 12
1	0.443 171 73	0.444 232 38	0.442 677 02
2	0.444 028 86	0.445 088 99	0.443 552 25
5	0.443 993 77	0.445 053 93	0.443 517 60
6	0.443 993 77	0.445 053 94	0.443 517 61
7	0.443 993 77	-	-

Table 4.8: Compact table of loss for each noise case for measurement set 2. Trained on 500 samples. Empty sections means it converged earlier. Full table in Appendix A.6

MS 2	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.306 645 11	0.309 534 47	0.309 731 25
1	0.220 095 34	0.221 476 21	0.223 449 06
2	0.234 998 85	0.236 758 66	0.238 454 06
16	0.231 988 33	0.233 656 26	0.235 384 19
17	0.231 988 33	0.233 656 26	0.235 384 19
18	-	-	0.235 384 19

Table 4.9: Table of loss for each noise case for measurement set 3. Trained on 500 samples. Full table in Appendix A.7

MS 3	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.315 881 16	0.323 639 56	0.321 830 29
1	0.236 753 16	0.242 948 59	0.238 896 90
2	0.248 362 13	0.254 708 36	0.251 202 53
12	0.246 571 44	0.252 907 56	0.249 311 13
13	0.246 571 44	0.252 907 56	0.249 311 12
14	0.246 571 44	0.252 907 56	0.249 311 12

Table 4.10: Table of loss for each noise case for measurement set 4. Trained on 500 samples. Full table in Appendix A.8

MS 4	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.325 895 35	0.328 418 53	0.326 000 07
1	0.431 559 37	0.434 641 19	0.432 815 80
2	0.452 267 95	0.455 418 71	0.454 053 32
9	0.457 144 60	0.460 305 96	0.459 147 87
10	0.457 144 67	0.460 306 03	0.459 147 94
11	0.457 144 68	0.460 306 04	0.459 147 96

On a pure RMSE basis, the errors increase for MS 2 and 3, overall equally so. The change was however low enough that one could say they compare very close to the 2500 sample trained models. Given the 5 times lower amount of samples, it could certainly be more feasible from a data collection point of view. MS1 on the other hand improved a little in all cases, which seems odd. MS4 however is quite different going to 500 samples. The first iteration is notably much worse, with a RMSE of double that of the 2500 model. However, the point of convergence was lower, like for MS1.

The main assumption why MS1 and MS4 misbehave is in the way that the training data was altered. When splitting up the dataset to weight some of the random valve splits closer to the optimal, with the lower amount of samples available, it's possible that the space far away from optimal grew less dense, leaving larger uncertainty. On the other hand, the data closer to the optimal may have been better distributed, giving better convergence. Since random points were generated, the distribution may have affected some of the result, but the overall difference seems rather large to be caused by only a worse random distribution.

Another interesting observation is how all datasets converged with less samples. This is assumed to be because of fewer training datapoints, meaning there is less neighbours weighting in on the predictions. With fewer datapoints, there is larger "dead zones" which means there is less chance of the next prediction to change significantly enough to be affected by a new datapoint. In the case of 2500 samples, once a prediction is made, and a valve input is given, the new measurement have different points nearby to slightly shift the prediction, which is likely why the convergence is slower with more samples.

Extrapolating from that assumption, it may be that MS2 does provides well correlated measurements, as there is several datapoints affecting the decision, and also giving the best results. MS 1 performed worse, and finished earliest with both 500 and 2500 samples, which may mean that fewer of the measurements provide good information. MS3 converged relatively fast however, and also showed good results. Case 2 with only test noise took the longest. Possibly caused by the higher uncertainty in the data, leaving less accurate predictions that shift the predictions more each iteration. However, this is just speculation, as similar behaviours is only

seen in MS3 with 2500 samples so there is not enough information to be any more certain of that.

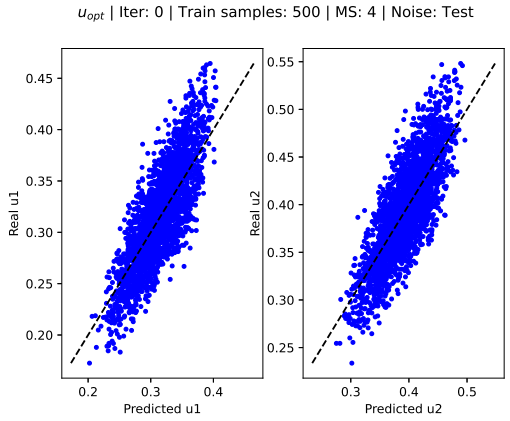

Figure 4.21: MS4: plot of the real values over the predicted values. Iteration 0. 500 samples used for training.

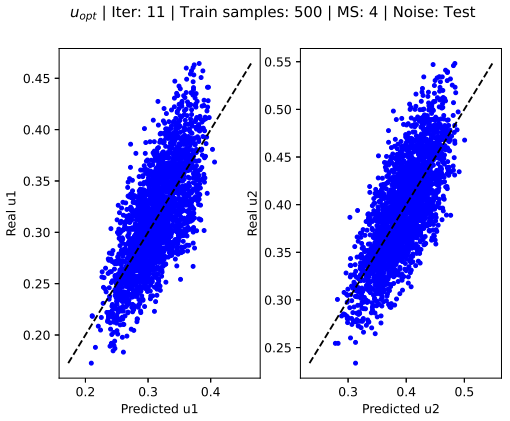

Figure 4.22: MS4: plot of the real values over the predicted values. Iteration 12. 500 samples used for training.

Unlike with 2500 samples, as seen in Table 4.10, MS4 converged when there was only 500 samples. The first iteration was much worse, but the convergence was not as bad compared to the version with more samples. MS4 was the least ”understandable” set, where the first prediction was much worse, as expected given less samples, but then the converged RMSE ended up lower than with more samples, similar to MS1. MS4 also converged relatively fast like MS1. It could be that the GP does not handle the random valve inputs as intuitively as a person would. With split data, which may act differently between 500 and 2500 samples, the performance was still odd. When the number of samples far away from the optimal was reduced, it led to divergence and a subsequent crash of the code in some cases for MS4, leading to having a relatively large chunk of information far away from the optimal.

5 Discussion

5.1 Convergence

Something that is seen throughout the measurement sets is that convergence is consistent, but it does not behave as ideally as one would wish. After the second iteration, the loss goes back up a little, This is non-ideal, despite the overall RMSE. The main assumption to this is that either there is not enough datapoints close to the optimum, or that that perhaps prediction is as good as it can be when receiving near the same inputs. The next prediction moves the system such that the new measurements are in a new ”area” closer to another point around the optimal. So both *cycling convergence* and *gradual convergence* were observed.

With 2500 points and 7 measurements as in the input, (MS4 for example) there is still a relatively low amount of points to chart out the space of the process. This was why there was implemented a shifting of the random

valve openings, such that that much data as possible could be concentrated closer to the optimum, where the main area of operation usually is. However it may not be proper enough. To improve on this, using Design of Experiments (DoE) to provide a controlled sampling of the working space of the process may perhaps help remedy this issue or just increase the performance for a fixed sample count.

5.2 Noise

Overall the performance seems good for all measurement sets, but measurement set 1 and 4 were the worse of the bunch. While MS1 converged fast, it's results were less accurate than MS2 and MS3. The fact that MS2 didn't converge is not expected to be much of an issue, given the stability of the RMSE. In a real world case, the noisy measurements and potential disturbances from moment to moment may be larger than those small fluctuations, so a convergence to this accuracy is probably not a requirement. Since MS3 uses heat capacity, then it might be more practical to make use of MS2 in a real plant, since that relied only on the temperatures, and still has the best performance.

Case 1, without noise, demonstrated it worked well and it was further observed that performance did not significantly degrade with noisy measurements on the test dataset in Case 2. If there exists a proper model of a HX network to generate data, even without noise, GP may prove sufficient to optimize the operation of it. It is however observed that in Case 3, noisy training measurements reduced the RMSE and improved performance a little. As such, if available, real world data or noisy model data would be better to train the GP.

5.3 Further work

Due to the nature of the system being an unconstrained optimum, the ideal method of control is controlling the gradient to zero.[7] Gradient prediction was out of scope for this project, as predicting gradients is one thing, but showing how it performs in a real world is much harder to do without implementing an optimization system that simulates and optimizes according to this method. However, predicting the gradients with respects to the output may be more of interest given the non-ideal convergence observed in the closed-loop approaches. Future work could be implementing both gradient prediction and valve opening prediction and comparing the performance of the methods.

Heat loss was also neglected, and may prove to an issue in real world applications, especially if the training data does not take heat loss into account. It's suggested to test for that as well.

6 Conclusions

Based on the results, measurement set 2 seems the most ideal dataset to use for predicting the optimum valve split, providing the most accurate results. Measurement set 3 did an acceptable job as well, but relies on the heat capacity of the flows, which may be harder to know. The performance of the other datasets were not way off but risk having some outliers at up to 2°C away from the optimum for MS1, and up to 5°C for MS4.

However for MS2 and MS3, almost all measurements were below 1°C from their optimum, with over half there again being under 0.5°C. Which makes them interesting candidates for optimization. It was observed that noise on test and training data affected performance a little, but still keep results very close to the noise free case.

For number of samples, it was observed that measurement sets seemed to converge faster with less samples, at the cost of performance for MS2 and MS3, while MS1 and MS4 converged to a lower RMSE value, which seemed counter-logical, but is assumed to be caused by the way the random valve positions that defined the first measurements were weighted towards the optimal.

With the two sample sizes, a rough idea of maximum sample count can be estimated for the measurement sets that worked well. For the other two, the inconsistency in convergence accuracy makes them harder to predict. The worse RMSE scores however, means that it's probably easier to focus on MS2 and MS3. And given that MS2 works only off of temperature measurements, it would be the simplest and best set to apply in practice.

References

[1] Carl Edward Rasmussen Ounpraseuth, Songthip and Christopher K. I. Williams. Gaussian processes for machine learning. *Journal of the American Statistical Association*, 103, 03 2008. doi: 10.2307/27640057.

[2] ENERGY FACTS NORWAY. Energy by sector. `https://energifaktanorge.no/en/norsk-energibruk/energibruken-i-ulike-sektorer`, 2020.

[3] Google Trends. Trends on machine learning. `https://trends.google.com/trends/explore?date=all&q=machine%20learning`, 16.12.2020.

[4] Carl Edward Rasmussen. *Evaluation of Gaussian Processes and Other Methods for Non-Linear Regression*. PhD thesis, University of Toronto, CAN, 1997. AAINQ28300.

[5] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN 9780071154673. URL `https://books.google.no/books?id=EoYBngEACAAJ`.

[6] GPy. GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`, since 2012.

[7] Sigurd Skogestad Johannes Jschke. Optimal operation of heat exchanger networks with stream split: Only temperature measurements are required. *Computers & Chemical Engineering*, 70:35 – 49, 2014. ISSN 0098-1354. doi: https://doi.org/10.1016/j.compchemeng.2014.03.020. URL `http://www.sciencedirect.com/science/article/pii/S009813541400101X`. Manfred Morari Special Issue.

A All Data

A.1 Noise cases

A.1.1 2500 Samples

Trained on 2500 samples, and prediction done on 2500 samples.

Table A.1: Table of loss for each noise case for measurement set 1. Trained on 2500 samples. Missing values means convergence was detected earlier.

MS 1	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.456 699 40	0.458 425 95	0.457 342 56
1	0.465 558 24	0.467 608 42	0.465 708 90
2	0.466 581 35	0.468 651 27	0.466 634 73
3	0.466 641 79	0.468 713 14	0.466 688 30
4	0.466 646 47	0.468 717 91	0.466 692 09
5	0.466 646 80	0.468 718 25	0.466 692 34
6	0.466 646 82	0.468 718 27	0.466 692 36
7	0.466 646 82	0.468 718 27	0.466 692 36

Table A.2: Table of loss for each noise case for measurement set 2. Trained on 2500 samples. Iteration stopped after not reaching the tolerance for convergence.

MS 2	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.284 660 44	0.290 504 89	0.291 154 63
1	0.201 940 21	0.205 108 59	0.206 691 28
2	0.219 144 98	0.222 869 11	0.223 518 50
3	0.212 930 96	0.216 549 85	0.217 845 62
4	0.215 061 83	0.218 706 36	0.219 695 48
5	0.214 187 07	0.217 833 19	0.218 981 96
6	0.214 549 41	0.218 192 73	0.219 263 96
7	0.214 383 08	0.218 029 93	0.219 141 58
8	0.214 460 92	0.218 105 65	0.219 196 46
9	0.214 421 71	0.218 068 25	0.219 170 37
10	0.214 441 71	0.218 087 18	0.219 183 12
11	0.214 430 83	0.218 077 21	0.219 176 65
12	0.214 436 77	0.218 082 56	0.219 180 00
13	0.214 433 29	0.218 079 60	0.219 178 22
14	0.214 435 32	0.218 081 27	0.219 179 18
15	0.214 434 04	0.218 080 31	0.219 178 66
16	0.214 434 84	0.218 080 87	0.219 178 95
17	0.214 434 29	0.218 080 54	0.219 178 78
18	0.214 434 66	0.218 080 74	0.219 178 88
19	0.214 434 39	0.218 080 62	0.219 178 82

Table A.3: Table of loss for each noise case for measurement set 3. Trained on 2500 samples. Missing values means convergence was detected earlier. Case 2 stopped after not reaching the tolerance for convergence at 20 iterations.

MS 3	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.290 814 15	0.302 511 04	0.299 119 99
1	0.220 568 12	0.229 063 49	0.227 277 26
2	0.230 636 01	0.239 809 92	0.237 484 31
3	0.228 610 82	0.237 538 40	0.235 421 18
4	0.229 046 61	0.238 058 08	0.235 871 55
5	0.228 943 39	0.237 925 20	0.235 762 71
6	0.228 968 99	0.237 960 99	0.235 790 34
7	0.228 962 39	0.237 950 89	0.235 783 02
8	0.228 964 14	0.237 953 84	0.235 785 03
9	0.228 963 66	0.237 952 95	0.235 784 46
10	0.228 963 79	0.237 953 23	0.235 784 63
11	0.228 963 76	0.237 953 14	0.235 784 58
12	0.228 963 77	0.237 953 17	0.235 784 59
13	0.228 963 76	0.237 953 16	0.235 784 59
14	0.228 963 76	0.237 953 16	0.235 784 59
15	0.228 963 76	0.237 953 16	0.235 784 59
16	0.228 963 76	0.237 953 16	0.235 784 59
17	-	0.237 953 16	0.235 784 59
18	-	0.237 953 16	-
19	-	0.237 953 16	-

Table A.4: Table of loss for each noise case for measurement set 4. Trained on 2500 samples. Iteration stopped after not reaching the tolerance for convergence at 20 iterations.

MS 4	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.218 891 75	0.220 646 41	0.219 080 34
1	0.370 358 80	0.373 660 98	0.374 139 84
2	0.446 822 09	0.451 062 44	0.455 208 88
3	0.481 877 19	0.486 633 39	0.493 870 22
4	0.497 642 77	0.502 674 28	0.512 052 35
5	0.504 776 51	0.509 959 26	0.520 697 53
6	0.508 051 94	0.513 320 05	0.524 885 03
7	0.509 581 56	0.514 898 56	0.526 955 11
8	0.510 308 03	0.515 653 21	0.527 999 38
9	0.510 658 55	0.516 019 98	0.528 536 51
10	0.510 830 12	0.516 200 91	0.528 817 89
11	0.510 915 19	0.516 291 34	0.528 967 88
12	0.510 957 87	0.516 337 06	0.529 049 17
13	0.510 979 50	0.516 360 40	0.529 093 96
14	0.510 990 56	0.516 372 43	0.529 119 04
15	0.510 996 27	0.516 378 66	0.529 133 34
16	0.510 999 24	0.516 381 91	0.529 141 63
17	0.511 000 80	0.516 383 62	0.529 146 53
18	0.511 001 62	0.516 384 52	0.529 149 49
19	0.511 002 05	0.516 384 99	0.529 151 31

A.1.2 500 Samples

Trained on 500 samples, and prediction done on 2500 samples.

Table A.5: Table of loss for each noise case for measurement set 1. Trained on 500 samples. Missing values means convergence was detected earlier.

MS 1	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.466 118 29	0.467 188 02	0.467 142 12
1	0.443 171 73	0.444 232 38	0.442 677 02
2	0.444 028 86	0.445 088 99	0.443 552 25
3	0.443 992 05	0.445 052 21	0.443 515 98
4	0.443 993 86	0.445 054 02	0.443 517 69
5	0.443 993 77	0.445 053 93	0.443 517 60
6	0.443 993 77	0.445 053 94	0.443 517 61
7	0.443 993 77	-	-

Table A.6: Table of loss for each noise case for measurement set 2. Trained on 500 samples. Empty sections means it converged earlier.

MS 2	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.306 645 11	0.309 534 47	0.309 731 25
1	0.220 095 34	0.221 476 21	0.223 449 06
2	0.234 998 85	0.236 758 66	0.238 454 06
3	0.231 233 45	0.232 874 97	0.234 606 49
4	0.232 194 18	0.233 870 21	0.235 598 18
5	0.231 930 96	0.233 596 41	0.235 324 08
6	0.232 004 82	0.233 673 53	0.235 401 60
7	0.231 983 49	0.233 651 17	0.235 379 04
8	0.231 989 79	0.233 657 80	0.235 385 75
9	0.231 987 89	0.233 655 79	0.235 383 71
10	0.231 988 47	0.233 656 41	0.235 384 34
11	0.231 988 29	0.233 656 22	0.235 384 14
12	0.231 988 35	0.233 656 28	0.235 384 20
13	0.231 988 33	0.233 656 26	0.235 384 18
14	0.231 988 33	0.233 656 27	0.235 384 19
15	0.231 988 33	0.233 656 26	0.235 384 19
16	0.231 988 33	0.233 656 26	0.235 384 19
17	0.231 988 33	0.233 656 26	0.235 384 19
18	-	-	0.235 384 19

Table A.7: Table of loss for each noise case for measurement set 3. Trained on 500 samples.

MS 3	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.315 881 16	0.323 639 56	0.321 830 29
1	0.236 753 16	0.242 948 59	0.238 896 90
2	0.248 362 13	0.254 708 36	0.251 202 53
3	0.246 226 99	0.252 563 71	0.248 952 21
4	0.246 644 07	0.252 979 50	0.249 385 22
5	0.246 555 21	0.252 891 60	0.249 294 94
6	0.246 575 26	0.252 911 29	0.249 314 85
7	0.246 570 51	0.252 906 66	0.249 310 23
8	0.246 571 68	0.252 907 79	0.249 311 35
9	0.246 571 38	0.252 907 51	0.249 311 07
10	0.246 571 46	0.252 907 58	0.249 311 14
11	0.246 571 44	0.252 907 56	0.249 311 12
12	0.246 571 44	0.252 907 56	0.249 311 13
13	0.246 571 44	0.252 907 56	0.249 311 12
14	0.246 571 44	0.252 907 56	0.249 311 12

Table A.8: Table of loss for each noise case for measurement set 4. Trained on 500 samples.

MS 4	Loss (RMSE)		
Iteration	1. No Noise	2. Test Noise	3. Train&Test
0	0.325 895 35	0.328 418 53	0.326 000 07
1	0.431 559 37	0.434 641 19	0.432 815 80
2	0.452 267 95	0.455 418 71	0.454 053 32
3	0.456 193 95	0.459 353 59	0.458 139 62
4	0.456 954 16	0.460 115 16	0.458 942 83
5	0.457 105 53	0.460 266 79	0.459 105 17
6	0.457 136 46	0.460 297 80	0.459 138 84
7	0.457 142 93	0.460 304 28	0.459 145 99
8	0.457 144 31	0.460 305 66	0.459 147 53
9	0.457 144 60	0.460 305 96	0.459 147 87
10	0.457 144 67	0.460 306 03	0.459 147 94
11	0.457 144 68	0.460 306 04	0.459 147 96

B Code

This part is divided into several files, and for practical purposed, it may be more convenient to have a look at the files directly.

B.1 g_process.py

This is the main GPy implementation, which deals with training and prediction.

```
import time
from multiprocessing import Process, Queue

import GPy
from sklearn.preprocessing import MinMaxScaler
import numpy as np


def predict_all(m: GPy.models.GPCoregionalizedRegression, X):
    ny = len(np.unique(m.output_index))
    y = []
    covy = []
    Xaug = np.hstack((X, 0.0 * np.ones_like(X[:, 0:1])))
    for iy in range(ny):
        Xaug[:, -1:] = iy
        y_i, covy_i = m.predict(Xaug, Y_metadata={'output_index': Xaug[:, -1:].astype(int)})
        y.append(y_i)
        covy.append(covy_i)
    return np.hstack(y), np.hstack(covy)


class GPModel(Process):
    def __init__(self):
        super(GPModel, self).__init__()
        self.train_queue = Queue()
        self.test_queue = Queue()
        self.output = Queue()

        self.norm_x = None
        self.norm_y = None
        self.normalize_y = False
        self.m = None

    def run(self) -> None:

        X_train, Y_train, num_restarts, normalize_y = self.train_queue.get()

        self.norm_x = MinMaxScaler((0, 1))
        X_train = self.norm_x.fit_transform(X_train)

        self.normalize_y = normalize_y
        if self.normalize_y:
            self.norm_y = MinMaxScaler((0, 1))
            Y_train = self.norm_y.fit_transform(Y_train)
            # print(norm_y.min_)
            # print(norm_y.scale_)
            # print(norm_y.feature_range)

        Y_train = np.array(list(zip(*Y_train)))
        Y_train = np.array([i[:, None] for i in Y_train])

        K = GPy.kern.RBF(input_dim=X_train.shape[1])
        icm = GPy.util.multioutput.ICM(input_dim=X_train.shape[1], num_outputs=Y_train.shape[1], kernel=K)
        np.random.seed(2311)
        self.m = GPy.models.GPCoregionalizedRegression([X_train, X_train], Y_train, kernel=icm)
```

```
        if num_restarts:
            self.m.optimize_restarts(messages=True, num_restarts=num_restarts)

        self.output.put(None)
        while True:
            X_test = self.test_queue.get()

            if X_test is None:
                return

            X_test = self.norm_x.transform(X_test)

            u_predicted, u_covariance = predict_all(self.m, X_test)

            if self.normalize_y:
                u_predicted = self.norm_y.inverse_transform(u_predicted)
                u_covariance /= self.norm_y.scale_ ** 2

            self.output.put((u_predicted, u_covariance, str(self.m)))

    def train(self, X_train, Y_train, num_restarts=0, normalize_y=False):
        self.train_queue.put((X_train, Y_train, num_restarts, normalize_y))
        return self.output.get()

    def test(self, X_test):
        self.test_queue.put(X_test)
        return self.output.get()

    def exit(self):
        self.test_queue.put(None)
```

B.2 u_optim_gp.py

```
import pprint
from copy import deepcopy

from optimal_u.hex3_gen_u_optim import load_data
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel

meas_sets = {
    1: ['T0', 'T1', 'T2', 'T3', 'Th1', 'Th2', 'Th3'],
    2: ['T0', 'Th1', 'Th2', 'Th3', 'The1', 'The2', 'The3'],
    3: ['T0', 'T', 'The1', 'The2', 'The3', 'w0', 'wh1', 'wh2', 'wh3'],
    4: ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha1', 'alpha2']
}


def generate_meas_set_data(mset, u, d):
    global meas_sets

    d = d.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u = np.array(list(zip(u['ur0'], u['ur1'])))

    results = []

    for idx, (u_, d_) in enumerate(zip(u, d)):
        # print(u_)
        # print(d_)
        result = hex3_chen.output_meas(mset, u_, d_)
        if not result['success']:
            print('Failed, small step in u')
            print(u_)

        res_dict = {}
        for idx, key in enumerate(meas_sets[mset]):
            res_dict[key] = result['y'][idx]
        results.append(res_dict)
    return np.array(results, dtype=dict)


def preprocess(u: pd.DataFrame, d: [pd.DataFrame, np.ndarray], d_order=None, u_input=None, add_noise=False):
    # Distubances
    if isinstance(d, pd.DataFrame):
        d = d.to_dict('records')

    # Optimal u values

    # This is u_chen
    if isinstance(u, pd.DataFrame):
        u_u0 = u['uc0']
        u_u1 = u['uc1']
    else:
        u_u0 = u[:, 0]
        u_u1 = u[:, 1]

    X = []

    # Ensure keys keep order
    if d_order is None:
        d_order = list(d[0].keys())
        # Remove the heat loss. Otherwise, random noise will be applied to 0 values, degrades performance.
        d_order.remove('h1')
        d_order.remove('h2')
```

```
        d_order.remove('h3')
        d_order.remove('Ts')

    # Random valve inputs
    if isinstance(u_input, pd.DataFrame):
        u_in0 = u_input['ur0']
        u_in1 = u_input['ur1']
    elif isinstance(u_input, np.ndarray):
        u_in0 = u_input[:, 0]
        u_in1 = u_input[:, 1]
    else:
        raise Exception('This_should_not_be_reached')
        u_in0 = u_u0
        u_in1 = u_u1

    d_keys = d_order.copy()
    toggle = 'alpha1' in d_keys

    if toggle:
        d_keys.remove('alpha1')
        d_keys.remove('alpha2')
    # Inputs

    np.random.seed(1)
    for idx, (u0, u1, d_row) in enumerate(zip(u_in0, u_in1, d)):
        x = [*([u0, u1] * toggle), *[d_row[k] + add_noise * np.random.normal(0, 1) for k in d_keys]]
        X.append(x)

    X = np.array(X)
    # Optimal u values
    Y = np.array(list(zip(u_u0, u_u1)))

    return X, Y, d_order


def plot_prediction(output, target, train_samples, noise, iters=0, MS=None):
    # Plotting

    fig, ax = plt.subplots(ncols=2)
    fig.suptitle(f"$u_{{opt}}$_|_Iter:_{iters}_|_"
                 f"Train_samples:_{train_samples}_{('|_MS:_'_+_str(MS))_*_bool(MS)}_|_Noise:_{noise}")

    ax[0].plot(output[:, 0], target[:, 0], 'b.')
    ax[0].plot([min(target[:, 0]), max(target[:, 0])], [min(target[:, 0]), max(target[:, 0])], 'k--')
    ax[0].set_xlabel('Predicted_u1')
    ax[0].set_ylabel('Real_u1')

    # ax[0].set_xlim([np.min(U_predicted-0.1), np.max(U_predicted)])
    # ax[0].set_ylim([np.min(U_test-0.1), np.max(U_test)])

    ax[1].plot(output[:, 1], target[:, 1], 'b.')
    ax[1].plot([min(target[:, 1]), max(target[:, 1])], [min(target[:, 1]), max(target[:, 1])], 'k--')
    ax[1].set_xlabel('Predicted_u2')
    ax[1].set_ylabel('Real_u2')
    # ax[1].set_xlim([np.min(U_predicted-0.1), np.max(U_predicted)])
    # ax[1].set_ylim([np.min(U_test-0.1), np.max(U_test)])
    plt.savefig(f'figs\\pred_u_{train_samples}_iter{iters}{("_MS"_+_str(MS))_*_bool(MS)}_{noise}.eps')
    plt.close()


def get_cost(output, target, params):
    predicted_cost = np.zeros(output.shape[0])
    real_cost = np.zeros(output.shape[0])

    params_d = params['d'].to_dict('records')
    params_J = params['J_chen']
    neg_u = 0
    large_u = 0
```

```
    for idx, (u_p, u_real, param) in enumerate(zip(output, target, params_d)):
        # print(params)
        if (u_p < 0).any():
            neg_u += 1

        if (np.sum(u_p) > 1).any():
            large_u += 1
        try:
            r = hex3_chen.cost(u_p, deepcopy(param))
            cost_p = - r['J']

            assert r['success']

            cost_r = params_J.iloc[idx]['J_chen']

            assert cost_p > 0
            assert cost_r > 0
        except AssertionError:
            return predicted_cost, real_cost

        predicted_cost[idx] = cost_p
        real_cost[idx] = cost_r

    print('Negative_u_count', neg_u)
    print('sum_U_over_1:', large_u)

    return predicted_cost, real_cost


def plot_loss(predicted, target, title, fp):
    # real_cost = test_data['J_chen'].values
    plt.plot(target, predicted, 'b.')
    plt.title(title)
    plt.ylabel('Predicted_cost_J')
    plt.xlabel('Real_cost_J')
    plt.plot([min(target), max(target)], [min(target), max(target)], 'k--')
    plt.savefig(f'figs\\{fp}.eps')
    plt.close()


def plot_hist(predicted, target, title, fp):
    # histogram
    abs_value_error = np.abs(predicted - target)
    print(min(abs_value_error))
    print(max(abs_value_error))
    plt.title(title)
    plt.hist(abs_value_error, bins=100)
    plt.ylabel('Number_of_error_points')
    plt.xlabel('Loss_|_Cost_Function')
    plt.savefig(f'figs\\{fp}.eps')
    plt.close()


def load_train_test(train, test, meas_set=None, closed_loop=False, train_noise=False, test_noise=False, norm_y=False):
    if train_noise and test_noise:
        noise = 'TrainTest'
    elif test_noise:
        noise = 'Test'
    elif train_noise:
        noise = 'Train' # Not tested for
    else:
        noise = 'None'

    test_data = load_data(test)
    train_data = load_data(train)

    if meas_set is not None:
```

```
    global meas_sets
    d_keys = meas_sets[meas_set]
    print('Solving_measurment_set_data...')
    ur = train_data['u_rand']
    uc = train_data['u_chen']
    u1 = np.array(list(zip(ur['ur0'], ur['ur1'])))
    u2 = np.array(list(zip(uc['uc0'], uc['uc1'])))
    diff = u2 - u1
    split = int(0.70 * len(u1))
    u3 = np.concatenate((u1[:split] + diff[:split] * 0.98, u1[split:] + diff[split:] * 0.5))
    # u3 = u1 + diff * 0.95

    tur = test_data['u_rand']
    tuc = test_data['u_chen']
    tu1 = np.array(list(zip(tur['ur0'], tur['ur1'])))
    tu2 = np.array(list(zip(tuc['uc0'], tuc['uc1'])))
    diff = tu2 - tu1
    tu3 = tu1 + diff * 0.95
    # u = np.concatenate([u1[:-(len(u1)*40) // 100], u2[-(len(u1)*40) // 100:]])
    test_u_input = tu3

    disturbances_train = generate_meas_set_data(meas_set, u3, train_data['d'])
    disturbances_test = generate_meas_set_data(meas_set, test_u_input, test_data['d'])
else:
    d_keys = None
    disturbances_train = train_data['d']
    disturbances_test = test_data['d']
    # Not used, dummy variable
    test_u_input = test_data['u_rand']
    u3 = train_data['u_rand']

X_train, U_train, key_order = preprocess(train_data['u_chen'], disturbances_train, d_order=d_keys,
                                         u_input=u3, add_noise=train_noise)
X_test, U_test, _ = preprocess(test_data['u_chen'], disturbances_test, key_order, u_input=test_u_input,
                               add_noise=test_noise)

# Hyperparameter
num_restarts = 1

mpr = GPModel()
mpr.start()
mpr.train(X_train, U_train, num_restarts, norm_y)
result = mpr.test(X_test)

U_predicted, U_covariance, m = result

target = U_test

plot_prediction(U_predicted, U_test, len(X_train), noise, iters=0, MS=meas_set)

iter_u = [U_predicted]

predicted_cost, real_cost = get_cost(U_predicted, target, test_data)

losses = get_errors(predicted_cost, real_cost)
pred_error = [get_errors(U_predicted, target)['RMSE']]
RMSE_loss_iter = [losses['RMSE']]

if closed_loop and meas_set is not None:
    i = 1
    while True:
        disturbances_test = generate_meas_set_data(meas_set, U_predicted, test_data['d'])

        X_test, _, _ = preprocess(test_data['u_chen'], disturbances_test, key_order, U_predicted,
                                  add_noise=test_noise)
        U_predicted, U_covariance, regression_info = mpr.test(X_test)

        predicted_cost, real_cost = get_cost(U_predicted, target, test_data)
```

```
            pred_error.append(get_errors(U_predicted, target)['RMSE'])
            losses = get_errors(predicted_cost, real_cost)
            RMSE_loss_iter.append(losses['RMSE'])

            iter_u.append(U_predicted)
            print(RMSE_loss_iter)
            if (np.abs(iter_u[-1] - iter_u[-2]) < 1e-8).all():
                break
            elif i == 19:
                print('Failed to converge')
                break
            i += 1

            # plot_prediction(U_predicted, U_test, len(X_train), iters=i, MS=meas_set)
        plot_prediction(U_predicted, U_test, len(X_train), noise, iters=i, MS=meas_set)

    losses = get_errors(U_predicted, target, U_covariance)

    MS = meas_set
    if MS is not None:
        plt.plot(list(range(len(iter_u))), RMSE_loss_iter)
        plt.title(f"ML Loss per iter | $u_{{opt}}$ | "
                  f"Train samples: {len(X_train)} {('| MS: ' + str(meas_set)) * bool(meas_set)} | Noise: {noise}")
        plt.ylabel('Loss - RMSE')
        plt.xlabel('Iteration')
        plt.savefig(f'figs\\iter_loss_{len(X_train)}_{("MS" + str(MS)) * bool(MS)}_{noise}.eps')
        plt.close()
    # print(*[u[0] for u in iter_u])
    # for i in [*[u[:5] for u in iter_u]]:
    # print(i)
    # print(U_test[:5])
    # print('-' * 20)
    title = f"Loss | $u_{{opt}}$ | Train_samples: {len(X_train)} " \
        f"{('| MS: ' + str(meas_set)) * bool(meas_set)} | Noise: {noise}"
    fp = f'loss_{len(X_train)}{("_MS" + str(MS)) * bool(MS)}_{noise}'
    plot_loss(predicted_cost, real_cost, title, fp)

    title = f"Histogram_loss | $u_{{opt}}$ | Train_samples: {len(X_train)} " \
        f"{('| MS: ' + str(meas_set)) * bool(meas_set)} | Noise: {noise}"
    fp = f'histloss_{len(X_train)}{("_MS" + str(MS)) * bool(MS)}_{noise}'
    plot_hist(predicted_cost, real_cost, title, fp)
    print('Pred_max', np.max(U_predicted))
    print('Pred_min', np.min(U_predicted))

    fp = f'data\\uopt_{len(X_train)}{("_MS" + str(MS)) * bool(MS)}_{noise}.csv'
    np.savetxt(fp, np.array(iter_u)[-1], delimiter=",")

    fp = f'data\\loss_{len(X_train)}{("_MS" + str(MS)) * bool(MS)}_{noise}.csv'
    np.savetxt(fp, np.array(RMSE_loss_iter), delimiter=",")

    mpr.exit()
    return losses, RMSE_loss_iter


def get_errors(predicted, target, cov=None):
    # Errors
    avg_dev = np.sum((predicted - target) ** 2) / len(target)
    avg_dev_norm = np.sum(((predicted - target) / target) ** 2) / len(target)

    errors = {'MSE': avg_dev, 'WMSE': avg_dev_norm, 'RMSE': np.sqrt(avg_dev)}

    if cov is not None:
        avg_dev_cov = np.sum((predicted - target) ** 2 / cov) / len(target)
        errors['CMSE'] = avg_dev_cov
    return errors
```

```
if __name__ == '__main__':
    samples = [500, 2500]


    # Uncomment this to show plots
    # plt.savefig = lambda x: plt.show()

    # uncommenth these two lines to save images as png
    # _savefig = plt.savefig
    # plt.savefig = lambda x: _savefig(x.replace('figs', 'imgs').replace('eps', 'png'))

    # Default saves as .eps files.

    test = '.\\datasets\\test_u_prediction2500.csv'
    import os

    if not os.path.isdir('results'):
        os.mkdir('results')

    # Black box
    pp = pprint.PrettyPrinter(indent=4)
    RESULTS = {}
    for MS in [1, 2, 3, 4]:
        RESULTS[MS] = {}
        for sample in samples:
            RESULTS[MS][sample] = {}
            train = f'.\\datasets\\train_u_prediction{sample}.csv'

            noise_label = 'No_measurment_noise'
            error, loss = load_train_test(train, test, meas_set=MS, closed_loop=True, train_noise=False,
                                                    test_noise=False)
            RESULTS[MS][sample][noise_label] = {'errors': error, 'RMSE_Loss': loss}

            for train_noise, noise_label in zip([True, False], ['With_train_noise', 'without_train_noise']):
                error, loss = load_train_test(train, test, meas_set=MS, closed_loop=True, train_noise=train_noise,
                                                        test_noise=True)
                RESULTS[MS][sample][noise_label] = {'errors': error, 'RMSE_Loss': loss,
                                                    }
            pp.pprint(RESULTS)

    pp.pprint(RESULTS)

    train = f'.\\datasets\\train_u_prediction{500}.csv'
    error, loss = load_train_test(train, test, meas_set=None, closed_loop=False, train_noise=False,
                                            test_noise=False)
    RESULTS[0] = {}
    RESULTS[0][500] = {}

    RESULTS[0][500]['No_measurment_noise'] = {'errors': error, 'RMSE_Loss': loss}

    for train_noise, noise_label in zip([True, False], ['With_train_noise', 'without_train_noise']):
        error, loss = load_train_test(train, test, meas_set=None, closed_loop=False, train_noise=train_noise,
                                                test_noise=True)
        RESULTS[0][500][noise_label] = {'errors': error, 'RMSE_Loss': loss}

    pp.pprint(RESULTS)

    # # Code for comparing normalization on output.
    # RESULTS[2] = {}
    # RESULTS[2][2500] = {}
    # train = f'.\\datasets\\train_u_prediction{2500}.csv'
    # for train_noise, noise_label in zip([True, False], ['normed_y', 'not_normed_y']):
    # error, loss, other = load_train_test(train, test, meas_set=2, closed_loop=True, train_noise=True,
    # test_noise=True, norm_y=train_noise)
    # RESULTS[2][2500][noise_label] = {'errors': error, 'MSE_Loss': loss}
    # pp.pprint(RESULTS)
```

B.3 hex3_gen_u_optim.py

```
import numpy as np
import hex3_old as hex3
import hex3_chen_old as hex3_chen
import copy
import pandas as pd


def gen_dataset(N, ttratio=1.0):
    # Smith, Noah A., and Roy W. Tromble. "Sampling uniformly from the unit simplex." Johns Hopkins University, Tech. Rep 29 (2004).
    dim = 3
    x = np.sort(np.random.rand(dim − 1, N ∗ 10), axis=0)
    x = np.concatenate([np.zeros((1, N ∗ 10)), x, np.ones((1, N ∗ 10))], axis=0)
    alpha = x[1:] − x[:−1]

    # # Checking uniformity
    # ax = plt.axes(projection='3d')
    # ax.plot(alpha[0], alpha[1], alpha[2] ,'b.')
    # plt.show()

    alpha = alpha[:−1]

    parspan = {}
    # Defining disturbance box [center, variability]
    parspan['T0'] = [60, 10] # C
    parspan['w0'] = [105, 25] # kW/K
    parspan['wh1'] = [40, 10] # kW/K
    parspan['wh2'] = [50, 10] # kW/K
    parspan['wh3'] = [30, 10] # kW/K
    parspan['Th1'] = [150, 30] # C
    parspan['Th2'] = [150, 30] # C
    parspan['Th3'] = [150, 30] # C
    parspan['UA1'] = [65, 15] # kW/K
    parspan['UA2'] = [80, 10] # kW/K
    parspan['UA3'] = [95, 15] # kW/K

    # Copied from transfer learning
    parspan['Ts'] = [0, 0] # C
    parspan['h1'] = [0, 0] # kW/K
    parspan['h2'] = [0, 0] # kW/K
    parspan['h3'] = [0, 0] # kW/K

    randmatrix = np.random.rand(len(parspan), N ∗ 10)
    parvec = {}
    for i, parname in enumerate(parspan.keys()):
        parvec[parname] = parspan[parname][0] + ttratio ∗ (2 ∗ randmatrix[i] − 1) ∗ (parspan[parname][−1])

    par0 = [{key: value[i] for key, value in parvec.items()} for i in range(N ∗ 10)]


    # Generating measurements, priors and targets
    u_span = []
    u_rand_span = []
    u_span_chen = []
    d_span = []
    J_span = []
    J_span_chen = []

    errors = 0
    finished = 0
    i = 0
    while finished < N:
        params = par0[i]

        if any(alpha[:, i] < 0.08) or sum(alpha[:, i] > 0.92):
            errors += 1
            i += 1
```

```
        print('Too low or high alphas: ', errors)
        continue

    u_chen = hex3_chen.optim(copy.deepcopy(params))
    u = hex3.optim(copy.deepcopy(params))
    # Calculate optimal output temp from optimal u
    if not u_chen['success'] or not u['success']:
        errors += 1
        i += 1
        print('Bad u_opt, errors: ', errors)
        continue

    cost = hex3.cost(u['u'], copy.deepcopy(params))
    cost_chen = hex3_chen.cost(u_chen['u'], copy.deepcopy(params))

    if not cost['success'] or not cost_chen['success']:
        errors += 1
        i += 1
        print('Bad hex cost solved, errors: ', errors)
        continue
    else:
        print('Success solutions: ', finished + 1)
        i += 1
        finished += 1

    # print(cost)
    # Save values
    u_span.append(np.array(u['u']))
    u_rand_span.append(alpha[:, i])
    u_span_chen.append(np.array(u_chen['u']))
    d_span.append(params)
    J_span.append(-cost['J'][0])
    J_span_chen.append(-cost_chen['J'][0])

    # For Scipy Implemntation (NOT USED)
    # params = par0[i]
    #
    # u = hex3_chen.optim(copy.deepcopy(params))
    # # Calculate optimal output temp from optimal u
    # if not u['success']:
    # errors += 1
    # i+=1
    # print('Bad u_opt, errors: ', errors)
    # continue
    # cost = hex3.cost(u['u'], copy.deepcopy(params))
    # cost_chen = hex3_chen.cost(u['u'], copy.deepcopy(params))
    # # gradient_chen = hex3_chen.grad(u, copy.deepcopy(params))
    # # grad = hex3.grad(u, copy.deepcopy(params))['grad']
    #
    # if not cost['success']:
    # errors += 1
    # i+=1
    # print('Bad hex cost solved, errors: ', errors)
    # continue
    # elif not cost_chen['success']:
    # errors += 1
    # i += 1
    # print('Bad hex chen cost solved, errors: ', errors)
    # continue
    # else:
    # print('Success solutions: ', finished)
    # i += 1
    # finished += 1

    # # Save values
    # u_span.append(np.array(u['u']))
    # d_span.append(params)
    # J_span.append(-cost['J'])
```

```
        # J_span_chen.append(-cost_chen['J'])

    u_span = np.array(u_span)
    u_rand_span = np.array(u_rand_span)
    u_span_chen = np.array(u_span_chen)
    d_span = np.array(d_span, dtype=dict)
    J_span = np.array(J_span)
    J_span_chen = np.array(J_span_chen)

    return u_span, u_span_chen, u_rand_span, d_span, J_span, J_span_chen


def save_data(name, u_span, u_span_chen, u_rand_span, d_span, J_span, J_span_chen):
    u_headers = [f'u{i}' for i in range(u_span.shape[1])]
    u_chen_headers = [f'uc{i}' for i in range(u_span_chen.shape[1])]
    u_rand_headers = [f'ur{i}' for i in range(u_rand_span.shape[1])]
    J_header = 'J'
    J_chen_header = 'J_chen'

    u_span_pd = pd.DataFrame.from_dict({key: val for key, val in zip(u_headers, u_span.T)})
    u_span_chen_pd = pd.DataFrame.from_dict({key: val for key, val in zip(u_chen_headers, u_span_chen.T)})
    u_rand_span_pd = pd.DataFrame.from_dict({key: val for key, val in zip(u_rand_headers, u_rand_span.T)})
    J_span_pd = pd.DataFrame.from_dict({J_header: J_span})
    # print(J_span)
    # print(J_span_chen)
    J_span_chen_pd = pd.DataFrame.from_dict({J_chen_header: J_span_chen})
    d_span_pd = pd.DataFrame.from_records(d_span)

    frames = pd.concat([u_span_pd, u_span_chen_pd, u_rand_span_pd, d_span_pd, J_span_pd, J_span_chen_pd], axis=1)

    frames.to_csv(name, index=False)


def load_data(name):
    frames = pd.read_csv(name)
    data = dict(u=frames.iloc[:, :2],
                u_chen=frames.iloc[:, 2:4],
                u_rand=frames.iloc[:, 4:6],
                d=frames.iloc[:, 6:-2],
                J=frames.iloc[:, -2:-1],
                J_chen=frames.iloc[:, -1:])
    return data


if __name__ == '__main__':

    # training sets
    for samples in [100, 500, 1000, 2500, 6000]:
        print('Generating_training_data....')

        np.random.seed(2030)
        u_span, u_span_chen, u_rand_span, d_span, J_span, J_span_chen = gen_dataset(samples, 1)
        save_data(f'.\\datasets\\train_u_prediction{samples}.csv', u_span, u_span_chen, u_rand_span, d_span, J_span,
            ↪ J_span_chen)
        print('Done')

    # Test set
    print('Generating_test_data....')
    np.random.seed(2028)
    samples = 2500
    u_span, u_span_chen, u_rand_span, d_span, J_span, J_span_chen = gen_dataset(samples, 1.2)
    save_data(f'.\\datasets\\test_u_prediction{samples}.csv', u_span, u_span_chen, u_rand_span, d_span, J_span, J_span_chen)
    print('Done')
```

B.4 hex3_chen_old.py

```
from casadi import *
import numpy as np

nlpopts = {'ipopt': {'print_level':0}, 'print_time':False};
x_vars = ['alpha3','T','Tstar1','Tstar2','Tstar3','The1','The2','The3','Q1','Q2','Q3','Qloss1','Qloss2','Qloss3','T1','T2','T3'];
u_vars = ['alpha1','alpha2']

meas_sets = {
    1: ['T0', 'T1', 'T2', 'T3', 'Th1', 'Th2', 'Th3'],
    2: ['T0', 'Th1', 'Th2', 'Th3', 'The1', 'The2', 'The3'],
    3: ['T0', 'T', 'The1', 'The2', 'The3', 'w0', 'wh1', 'wh2', 'wh3'],
    4: ['T0', 'T', 'Th1', 'Th2', 'Th3', 'alpha1', 'alpha2']
}

Ti_max = 1500


def model(par):
    T = SX.sym('T');
    Tstar1 = SX.sym('Tstar1');
    Tstar2 = SX.sym('Tstar2');
    Tstar3 = SX.sym('Tstar3');
    The1 = SX.sym('The1');
    The2 = SX.sym('The2');
    The3 = SX.sym('The3');
    Q1 = SX.sym('Q1');
    Q2 = SX.sym('Q2');
    Q3 = SX.sym('Q3');
    Qloss1 = SX.sym('Qloss1');
    Qloss2 = SX.sym('Qloss2');
    Qloss3 = SX.sym('Qloss3');
    T1 = SX.sym('T1');
    T2 = SX.sym('T2');
    T3 = SX.sym('T3');
    alpha1 = SX.sym('alpha1');
    alpha2 = SX.sym('alpha2');
    alpha3 = SX.sym('alpha3');

    T0 = par['T0'];
    w0 = par['w0'];
    Th1 = par['Th1'];
    Th2 = par['Th2'];
    Th3 = par['Th3'];
    wh1 = par['wh1'];
    wh2 = par['wh2'];
    wh3 = par['wh3'];
    UA1 = par['UA1'];
    UA2 = par['UA2'];
    UA3 = par['UA3'];

    Ts = par['Ts'];
    h1 = par['h1'];
    h2 = par['h2'];
    h3 = par['h3'];

    dTlm1 = ( (Th1 - Tstar1) * (The1 - T0) * ( (Th1 - Tstar1) + (The1 - T0) )/2)**(1/3);
    dTlm2 = ( (Th2 - Tstar2) * (The2 - T0) * ( (Th2 - Tstar2) + (The2 - T0) )/2)**(1/3);
    dTlm3 = ( (Th3 - Tstar3) * (The3 - T0) * ( (Th3 - Tstar3) + (The3 - T0) )/2)**(1/3);

    f0 = - T + alpha1*T1 + alpha2*T2 + alpha3*T3;
    f01 = alpha1 + alpha2 + alpha3 - 1;
    f11 = - Q1 + w0*alpha1*(Tstar1 - T0);
    f12 = - Q2 + w0*alpha2*(Tstar2 - T0);
    f13 = - Q3 + w0*alpha3*(Tstar3 - T0);
    f21 = - Q1 + UA1*dTlm1;
    f22 = - Q2 + UA2*dTlm2;
    f23 = - Q3 + UA3*dTlm3;
```

```
    f31 = - Q1 + wh1*(Th1 - The1);
    f32 = - Q2 + wh2*(Th2 - The2);
    f33 = - Q3 + wh3*(Th3 - The3);
    f41 = - Qloss1 + w0*alpha1*(T1 - Tstar1);
    f42 = - Qloss2 + w0*alpha2*(T2 - Tstar2);
    f43 = - Qloss3 + w0*alpha3*(T3 - Tstar3);
    f51 = - Qloss1 + h1*(Ts - T1);
    f52 = - Qloss2 + h2*(Ts - T2);
    f53 = - Qloss3 + h3*(Ts - T3);

    x = vertcat(alpha3,T,Tstar1,Tstar2,Tstar3,The1,The2,The3,Q1,Q2,Q3,Qloss1,Qloss2,Qloss3,T1,T2,T3);
    f = vertcat(f0,f01,f11,f12,f13,f21,f22,f23,f31,f32,f33,f41,f42,f43,f51,f52,f53);
    u = vertcat(alpha1,alpha2);
    J = -T;

    return {'x': x, 'u': u, 'f': f, 'J': J}

def output(u, par, x0=None):

    m = model(par);
    nx = np.prod(m['x'].shape);
    nu = np.prod(m['u'].shape);
    nf = np.prod(m['f'].shape);

    if x0 is None:
        x0 = np.array( [0.33]*(nu+1) + [(par['T0']+par['Th1'])/2, (par['T0']+par['Th2'])/2, (par['T0']+par['Th3'])/2]*2 + [
            ↪ par['T0']] * (nx - 2*(nu+1) - 1) )

    nlp = {} # NLP declaration
    nlp['x'] = vertcat(m['u'],m['x']) # decision vars
    nlp['f'] = m['J'] # objective
    nlp['g'] = m['f'] # constraints

    # Create solver instance
    F = nlpsol('F','ipopt',nlp,nlpopts);

    # Solve the problem using a guess
    lbx = np.array([*u]+[0]*(1)+[-inf]*(nx-1)); # constraint on inputs and first state (last flow split)
    ubx = np.array([*u]+[1]*(1)+[+inf]*(nx-1)); # upper limit on splits is not necessary, but will automatically be satisfied

    lbx[(nu+2):(nu+2+3*2)] = par['T0']; # constraint on temperatures
    ubx[(nu+2):(nu+2+3*2)] = np.array([par['Th1'], par['Th2'], par['Th3'], par['Th1'], par['Th2'], par['Th3']]); # constraint
        ↪ on temperatures


    r = F(x0=x0, lbg=np.zeros(nf), ubg=np.zeros(nf), lbx=lbx, ubx=ubx);

    sol = r['x'].full().reshape(-1)

    return {'x': sol[-nx:], 'success': F.stats()['success']}

def cost(u,par):
    m = model(par);
    F = Function('F',[m['x'], m['u']],[m['J']], ['x','u'], ['J']);
    out = output(u,par); #np.zeros(nx);
    J = F(out['x'], u);
    return {'J': J.full().reshape(-1), 'success': out['success']};

def grad(u,par):
    m = model(par);

    F = Function('F',[m['x'], m['u']],[m['f'], m['J']], ['x','u'], ['f','J']);
    G = rootfinder('G','newton',F)
    out = output(u,par); #np.zeros(nx);
    Jufun = G.factory('Ju', ['x','u'], ['jac:J:u']);

    delta = 0;
    Ju = Jufun(out['x']+delta, u).full().reshape(-1)
```

```
    # while not G.stats()['success']:
    # delta = delta*10;
    # Ju = Jufun(xguess+delta, u).full().reshape(-1)
    return {'grad': Ju, 'success': True};

def output_meas(meas_set, u, par):
    meas_vars = meas_sets[meas_set];
    y = np.zeros((len(meas_vars),));
    out = output(u, par);
    x = out['x'];
    for i, var in enumerate(meas_vars):
        if var in par:
            y[i] = par[var];
        elif var in u_vars:
            y[i] = u[u_vars.index(var)];
        elif var in x_vars:
            y[i] = x[x_vars.index(var)];
        else:
            y[i] = np.nan;
    return {'y': y, 'success': out['success']}

def optim(par, x0=None):

    m = model(par);
    nx = np.prod(m['x'].shape);
    nu = np.prod(m['u'].shape);
    nf = np.prod(m['f'].shape);

    nlp = {} # NLP declaration
    nlp['x'] = vertcat(m['u'],m['x']) # decision vars
    nlp['f'] = m['J'] # objective
    nlp['g'] = m['f'] # constraints

    # Create solver instance
    F = nlpsol('F','ipopt',nlp,nlpopts);

    Tbackoff = 1;
    # Trand = 1;
    alphabackoff = 1e-3;

    if x0 is None:
        # x0 = np.zeros(nx+nu);
        # x0[:nu+1] = 1/(nu+1);
        # x0[(nu+2):(nu+2+3*2)] = par['T0'] + Tbackoff #+ Trand*np.random.rand(3*2); # [Tstar1,Tstar2,Tstar3,The1,
            ↪ The2,The3]
        x0 = np.array( [0.33]*(nu+1) + [(par['T0']+par['Th1'])/2, (par['T0']+par['Th2'])/2, (par['T0']+par['Th3'])/2]*2 + [
            ↪ par['T0']] * (nx - 2*(nu+1) - 1) )

    # Solve the problem using first guess

    lbx = np.array([alphabackoff]*(nu+1)+[-inf]*(nx-1)); # constraint on inputs and first state (last flow split)
    ubx = np.array([1]*(nu+1)+[+inf]*(nx-1)); # upper limit on splits is not necessary, but will automatically be satisfied

    lbx[(nu+2):(nu+2+3*2)] = par['T0']; # constraint on temperatures
    ubx[(nu + 2):(nu + 2 + 3)] = np.array([min(Ti_max, t) for t in [par['Th1'], par['Th2'], par['Th3']] ]); # constraint on
        ↪ temperatures
    ubx[(nu + 2 + 3):(nu + 2 + 3 * 2)] = np.array([par['Th1'], par['Th2'], par['Th3']]); # constraint on temperatures

    r = F(x0=x0, lbg=np.zeros(nf), ubg=np.zeros(nf), lbx=lbx, ubx=ubx);

    # while not F.stats()['success']:
    # Trand = Trand + 1;
    # x0[(nu+2):(nu+2+3*2)] = par['T0'] + Tbackoff + Trand*np.random.rand(3*2);
    # r = F(x0=x0, lbg=np.zeros(nf), ubg=np.zeros(nf), lbx=lbx, ubx=ubx);

    sol = r['x'].full().reshape(-1)

    return {'u': sol[:nu], 'x': sol[-nx:], 'success': F.stats()['success']}
```