Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Thomas Edvardsen

Optimization of heat exchanger networks using Gaussian process regression

Process control using Gaussian processes for near-optimal operation in the presence of active constraints

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Sigurd Skogestad
Co-supervisor: Lucas Ferreira Bernardino
June 2021

NTNU
Norwegian University of Science and Technology

Thomas Edvardsen

Optimization of heat exchanger networks using Gaussian process regression

Process control using Gaussian processes for near-optimal operation in the presence of active constraints

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Sigurd Skogestad
Co-supervisor: Lucas Ferreira Bernardino
June 2021

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

Optimal operation of heat exchanger networks can save energy and costs. This study investigated if Gaussian processes could be used to control the valve splits of a heat exchanger network. In practice it is hard to measure all the things needed for a full model based approach, and thus there is a need to work with a reduced set of measurements and aim for near-optimal performance instead. For heat exchangers, the temperatures are such well behaved measurements. Using the Gaussian process to predict gradients or optimal valve splits from different measurement sets, acceptable performance could be achieved, even in the presence of active temperature constraints. For purely maximizing the output temperature, predicting gradients and then using a setpoint controller worked best. For a temperature constrained case, a constrained surrogate controller predicting valve openings worked the best. In general, the gradient control structures reacted to changes in more disturbances than the surrogate controller, such as changes in heat capacity and the overall heat transfer coefficients. The surrogate controllers were more sensitive to the amount of samples near the optimum in the training dataset, where some measurements performed better with more. The constrained gradient control structure was better at staying at or below the constraint, but had subpar performance when below it. The constrained surrogate controller was also strongly affected by the measurement sets used, where a bad selection of measurements could cause divergence issues. The constrained mixed controller was the most stable performing control structure, with on average good performance, but not the best. The measurement sets that performed best was the ones containing variables using in the Jäschke temperature, which could be an indicator to the types of measurements that are ideal for prediction tasks such as this case.

Preface

This is a Master thesis building upon the work done as part of a specialization project performed the prior semester to starting work on the thesis. Continuing on from the specialisation project, the supervisor was Sigurd Skogestad and co-supervisor was Lucas Ferreira Bernardino. Their continued assistance is what made this work possible, and I cannot thank them enough.

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	Chen's approximation of logarithmic mean temperature difference	[°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.	[-]
UA	Overall heat transfer coefficient	[W/K]
Q	Heat transfer	[kW]

Contents

1 Introduction

Heat transfer and recovery is used in all kinds of industry, and on a large scale, heating associated costs can grow significant. Systems of heat exchangers need to be operated efficiently, while rejecting disturbances. Reducing the heat loss is good for both the company economics, as well as greener for the world.

It is not always so easy to find the optimal operating point of a system of heat exchangers, heat capacity and flows might not be easily measured or accurately predicted for example. Despite perhaps accurate models, without the accurate knowledge of the disturbances on the systems, acquiring accurate predictions can be complicated. The idea is then to try to use the variables that can be easily measured, such as temperatures, to estimate the optimal operating point of a system. The immediate problem however is that you are given less information than what is required to model the system precisely. And it can be very hard to simplify a model to only use variables that are easy to measure, while still keeping sufficient accuracy.

The potential solution is machine learning. Machine learning encompasses several methodologies, but one thing that makes it so powerful is that it can learn from data, and learn the underlying models from that data. So where accurate modelling is not feasible, machine learning may be a solution that provides sufficient approximations.

There are several machine learning methods, the most popular being neural network based ones. A neural network model is set up, and weights in the model layers are trained on data. If the network is properly set up and trained, it can learn the underlying hidden model or correlation in the training data. The model can then be used to make accurate predictions on new data. There are some potential downsides to this, as it is hard to completely understand what the neural networks learn, and thus makes predictions on.

Another machine learning method is based on Gaussian processes (GP), which uses covariance functions to measure correlation in the data, and then make predictions. Along with predictions, the variance is returned as well, giving a measure of how ”certain” the process is of its prediction. Overfitting is a risk in machine learning, especially within neural networks. Overfitting is the case where it does not learn the hidden rules or model of the data you give it, but instead memorizes the input-output as part of the network, leading to bad predictions when the network is used for real world predictions that are not part of the training dataset.

Gaussian processes can suffer from overfitting, from improperly chosen hyperparameters, but have the advantage that it will always respect the training data. Neural networks often require quite large training datasets to help it learn the underlying model in the data, and not memorize it. A validation or test dataset, which the network is not trained on, is used to verify that the model can do real world predictions correctly. For Gaussian processes training and validation are done using the same dataset by using statistical metrics. If the hyperparameters are well tuned, then the risk of overfitting is much smaller compared to traditional neural networks. A test dataset is still used here to measure performance on different data than the training data. In this project, the test data presents points in a larger range than the training data to measure how well it

could extrapolate beyond the training region.

The aim is thus to apply this Gaussian process to make predictions on a system, using a control structure where GP is a central component. By having the Gaussian process predict some variable from a selection of measurements of the system, sufficient process control may be achieved.

1.1 Scope of work

The focus of the work is to find out if Gaussian process regression can model a heat exchanger network with three heat exchangers in parallel, with a single input and a single output. The single input is split into the three heat exchangers. The goal is to control that split configuration based on the measured state of the system, to reach the optimal output temperature of the output stream. A model of the heat exchanger network is implemented and simulated, and a trained Gaussian process will be used to control the system. The goal is to see what measurement and what predictions the controller can use to gain the best performance.

A selection of measurement sets are tested to see what input data gives a good performance, and various predicted outputs are measured, such as predicting the gradient of the splits with respect to the inputs, or the optimal input configuration directly.

In this project, there are two working cases considered. The first case is unconstrained optimization, where the single goal is to maximize the output temperature. In this case two controller structures are implemented, one which performs surrogate optimization where it directly predicts the optimal configuration to get the highest output temperature. The other control structure predicts the gradients of the streams with respect to the valve openings, and thus acts closer to a traditional gradient based process controller.

The other case is where the system is constrained. Here the output temperature of each heat exchanger is constrained to a maximum operating temperature, and the goal is to investigate if GP methods still perform desirably under different operating criteria. The controllers from the unconstrained case are modified for this case, and a third controller is introduced, where elements of both the surrogate controller and the gradient based controller are combined.

2 Theory

2.1 Heat Exchanger Network

The cases in this work consider unconstrained and constrained optimization with the goal of maximizing the output temperature from a heat exchanger (HX) network. In this report, all temperatures mentioned are in °C.

An illustration of the setup is shown in Figure 2.1. Disturbances are the variables that along with the valve split that determine the state of the system. All disturbances are required to calculate the output temperature using a numerical model, denoted as the the plant in this report. α and β are also required to solve the model, and are considered system inputs to be manipulated. The T_i's describe the temperatures of the streams out of the heat exchangers, and T is the output temperature after the streams are merged. 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. For this case, w is the product of the stream flow and the specific heat capacity of the fluid. 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 refers to the hot stream going into the HX, and $_{\mathrm{he,i}}$ refers to the hot stream going out of that HX.

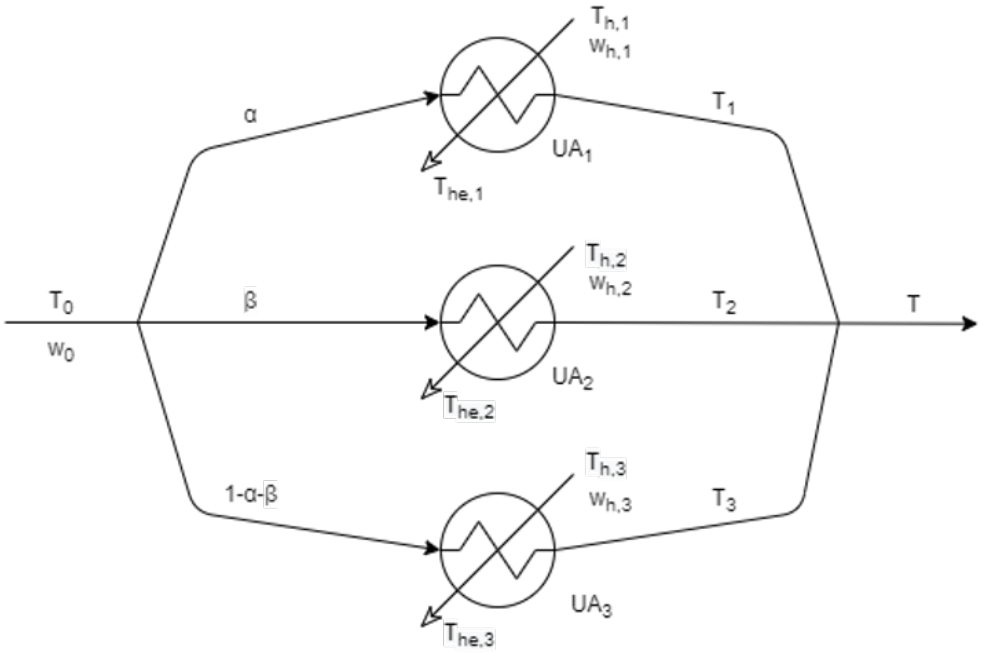


Figure 2.1: Illustration of the heat exchanger network. A input stream is split according to the values α and β, which are the valve splits. Each stream is heated through a heat exchanger before merged back into a single stream.

2.1.1 Model equations

This section details the equations in the modelling of the heat exchanger network. In Equation 2.1 the equality constraint of the temperature for the output stream is shown. Equation 2.2 details the equality constraint on the valve openings, where the γ is the remaining valve opening, however since it is determined by the other two, it is not considered worth including outside the numerical model implementation.

$$\mathrm{T} = \mathrm{T}_1 \cdot \alpha + \mathrm{T}_2 \cdot \beta + \mathrm{T}_3 \cdot \gamma \tag{2.1}$$

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

Equation 2.3 define Chen's approximation to the logarithmic mean temperature difference for each of the split streams. This approximation is used to avoid numerical problems with the model. [2]

$$\mathrm{dTlm_i} = \left((\mathrm{T_{h,i}} - \mathrm{T_i}) \cdot (\mathrm{T_{he,i}} - \mathrm{T_0}) \cdot \frac{1}{2} ((\mathrm{T_{h,i}} - \mathrm{T_i}) + (\mathrm{T_{he,i}} - \mathrm{T_0})) \right)^{\frac{1}{3}} \tag{2.3}$$

Equations 2.4 to 2.12 describe the heat transfer between hot and cold streams.

$$\mathrm{Q}_1 = \mathrm{w}_0 \cdot \alpha \cdot (\mathrm{T}_1 - \mathrm{T}_0) \tag{2.4}$$

$$\mathrm{Q}_2 = \mathrm{w}_0 \cdot \beta \cdot (\mathrm{T}_2 - \mathrm{T}_0) \tag{2.5}$$

$$\mathrm{Q}_3 = \mathrm{w}_0 \cdot \gamma \cdot (\mathrm{T}_3 - \mathrm{T}_0) \tag{2.6}$$

$$\mathrm{Q}_1 = \mathrm{w}_{\mathrm{h},1} \cdot \alpha \cdot (\mathrm{T}_{\mathrm{h},1} - \mathrm{T}_{\mathrm{he},1}) \tag{2.7}$$

$$\mathrm{Q}_2 = \mathrm{w}_{\mathrm{h},2} \cdot \beta \cdot (\mathrm{T}_{\mathrm{h},2} - \mathrm{T}_{\mathrm{he},2}) \tag{2.8}$$

$$\mathrm{Q}_3 = \mathrm{w}_{\mathrm{h},3} \cdot \gamma \cdot (\mathrm{T}_{\mathrm{h},3} - \mathrm{T}_{\mathrm{he},3}) \tag{2.9}$$

$$\mathrm{Q}_1 = \mathrm{UA}_1 \cdot \mathrm{dTLm1} \tag{2.10}$$

$$\mathrm{Q}_2 = \mathrm{UA}_2 \cdot \mathrm{dTLm2} \tag{2.11}$$

$$\mathrm{Q}_3 = \mathrm{UA}_3 \cdot \mathrm{dTLm3} \tag{2.12}$$

2.2 Surrogate optimization

Instead of accurately modelling the system, the aim of surrogate optimization techniques is to find local or global optima for operation, by approximating the optimization problem. In this case, there is just a selection of available measurements from the system which alone are not sufficient to model the system with a first-principle model. By using machine learning, a sufficient approximation may be possible.

Surrogate optimization is data driven, through random or controlled sampling of the design or operating space of the process. For this, an accurate model with all disturbances of the system can be used to simulate the process to generate data, or a real process could be sampled. In practice however, this is not that easy. For example, the heat capacity used in this projects model is a product of the mass flow and the specific

heat capacity. Uneven flows can be hard to estimate, and the specific heat capacity may vary with the composition of the stream, which makes heat capacity harder to measure accurately. On the other hand, measuring temperatures is much simpler and less error prone. Ideally, the machine learning models should only be trained on measurements that are well behaved. [3]

Data driven methods are likely not as accurate as model-based approaches, but can have the advantage of not needing the model and all its disturbances, as long as the machine learning prediction is good enough. The advantage is that creating a simplified model of the process is left to the data driven method at hand, such as machine learning. This saves time and effort spent on modelling, and if the measurement set were to change, a new model does not have to be made. Instead, new measurements need to be taken to re-train with the specific method used. If an accurate model of the system is available, then generating new measurements can be fast and easy.

2.3 Machine Learning

Machine learning is the method of having machines learn through experiencing the data. [4] It has been applied in several fields in modern times, and research is happening on even more. It exists in most peoples daily life in some form, like solving which advertisements you would be interested in. In machine learning, the model is trained on data which it is supposed to learn the underlying rules of. When some new input is given, it should be able to apply those rules to make a good estimate of the output. An example is spam filtering for emails, where it trains on known emails labeled spam or legitimate, and then when a new email is obtained, it can make a prediction if it is spam or not. This is an example of classification problems, where the output is discrete, a yes or no in this case. The counterpart, where the output is not discrete, is called regression.

Regression problems are special in the sense that they do no longer give a answer like yes or no, but instead make an output which may be a ”creation”. For example in a computer vision task, machine learning has been used to do image or video in-painting. Where a section of a video or image is removed, and the machine learning model is tasked with filling in the missing information, so that the image or video looks natural. In those cases, it has to learn the scene buildup and fill in what would be in the missing part. It will not be the original video, but it can be something that approximates a real video.

When training these machine learning models where they need to learn the rules of the system, it is important to have a dataset that reflects the type of data it should be able to predict, and cover most if not all the input space, so the model does not create unexpected results for outliers, datapoints far away from the dataset. While not such a big problem for a Gaussian processes, in neural networks it is often harder to know how some relatively unknown piece of information is going to be treated. This is why having a set of training data and a set of test or validation data is common, where you train on the former, and then validate with the latter. The validation data may contain more outliers and extreme inputs for the model, so as to make sure it performs sufficiently well on data that is slightly outside what it is trained on. It is also useful for

checking if a model does not overfit, as prediction performance on the output would visibly drop if the model just "memorized" the training data.

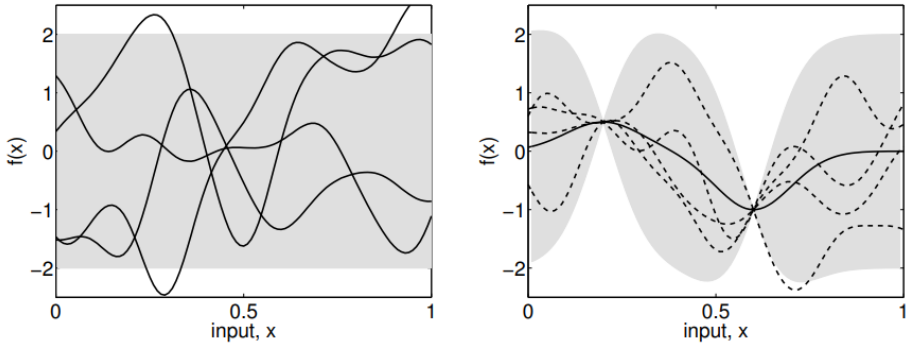
Machine learning methods can be divided into parametric and non-parametric. Parametric machine learning present a set of weights that are tuned through the training process and are core to the prediction, where the model with just different weights can perform different tasks. Artificial Neural Nets (ANNs) are parametric, and have hidden layers that have nodes, trying to emulate neural connections in the brain. These connections are tuned by weights that decided if the neuron is supposed to activate or not, depending on the input. A non-parametric model, such as Gaussian processes, does however not have such weights that need to be trained and put more focus on the hyperparameters. Hyperparameters are in both parametric and non-parametric machine learning, and can be considered what the initial "configuration" of the machine learning model is, such as choice of learning rate, kernel, or setting parameters connected to the specific method.

2.4 Gaussian Processes

The Gaussian process (GP) framework is based on supervised learning, where 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 train 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 later on.

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 is thus composed of the following "observations", $\mathcal{D} = \{(x_i, y_i)|i = 1, ..., n\}$, where n is the number of samples. The approach for establishing an input-output relationship over $\mathcal{D}$ is based on creating 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 is not easy to do as there can be infinite sets of possible functions to use. The Gaussian process 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]

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 functions 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_*.



(a), prior (b), posterior

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]

2.4.1 Kernel

The way data is connected in GP is through the covariance functions that describe the covariance between the datapoints, and thus the choice of function directly affects the nature of the data you want to make predictions from. The covariance function is also known as the kernel. The kernel be described as the dot product in a feature space, which is what GP predictions operate in. Within GPy, the python framework used, you can have the kernel be a sum of covariance 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 2.13. [1]

$$k_y(x_p, x_q) = \sigma_f^2 \exp\big(-\frac{1}{2 \cdot \ell^2}(x_p - x_q)^2\big) + \sigma_n^2 \delta_{pq} \tag{2.13}$$

Where the kernel in this case is referred to as k_y, where 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 GP formulation

The core of GP predictions lies in Equation 2.14. The equation incorporates both the training data, test input(s) to be inferred, and the kernel which describes the relation between datapoints to give a mean prediction(s) of the test input(s). $\mathbf{k}_*$ is a matrix containing the covariances between test datapoints and the n training datapoints found using the kernel. For a single test input $\mathbf{x}_*$, $\mathbf{k}_*$ would be a vector of length

n containing covariances between the test input and each training input, which are found with the kernel. This can be written as $\mathbf{k}_* = [k(\mathrm{x}_1, \mathbf{x}_*), k(\mathrm{x}_2, \mathbf{x}_*), ..., k(\mathrm{x}_n, \mathbf{x}_*)]^\mathsf{T}$, where k is the kernel function and x_i is a training input where i is the index of the training input. Similarly to $\mathbf{k}_*$, K is the matrix with covariances between all the training inputs, I is the identity matrix, and σ_n^2 is the variance of the noise of the system. Finally, $\mathbf{y}$ is the vector of training outputs. Equation 2.14 is a linear combination of observations $\mathbf{y}$, which can be referred to as a linear predictor. [1]

$$f_* = \mathbf{k}_*(\mathrm{K} + \sigma_n^2\mathrm{I})^{-1}\mathbf{y} \tag{2.14}$$

As mentioned, a strength of GP is that the variance is calculated as well for a prediction. Equation 2.15 describes how the variance is calculated, using many of the same terms as Equation 2.14, but here the first part of the equation is the covariance of the test input when compared to itself. As shown by the RBF kernel function in Equation 2.13, even if the inputs are the same, the term is still affected by the noise hyper-parameters, as signal noise and input noise are still factors in the computation. This comes from that the test input, or any input, may have noise, meaning it is not guaranteed to be an accurate measurement of the input we want to make a prediction for. This is interesting in the sense that these parameters can make predictions that somewhat accounts for noise in the inputs.

$$\mathbb{V}[f_*] = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^\mathsf{T}(\mathrm{K} + \sigma_n^2\mathrm{I})^{-1}\mathbf{k}_* \tag{2.15}$$

2.5 Self-optimizing control

The basic idea of self optimizing control is making a feedback optimizing control structure where the objective is to translate economic objectives into process objectives. Self-optimizing control (SOC) is about optimizing parts of a process which may not be necessary for process stability. For example, running heat exchanger networks for optimal heat recovery, reducing costs involved with heating up later. Taking a model based approach has some issues however, as creating an accurate model of what you want to optimize may be hard, and even harder to implement into the process.

The largest issue however, is similar to one in surrogate optimization. Specifically, measurement error and noise can be significant issues. Therefore, there is a need to simplify the implementation to a point where we do not target perfect performance, but near optimal, where the loss is acceptable. Selecting some controlled variables to get acceptable loss with a constant setpoint c_s, is when we have self-optimized control. The goal of SOC is to find a function, c, of the available measurements, y, such that when kept constant leads to near-optimal operating conditions. The problem is often more about selecting those variables to avoid issues with respect to disturbances, measurement error and noise while still having good enough performance. [5]

Optimization based on gradients is theoretically best for the noiseless case, as gradient based control with a constant setpoint of zero will always lead to optimal operation. The gradients are the cost function with respect to the manipulated variables. There is however a big problem with gradients, they too are very hard to find in practice, as they can not be measured directly. Gradients rely on the process model to be solved. So we run into the problem of needing a very good model and measurements to find those gradients. However, if a method can sufficiently estimate the gradients, then that may be good enough. This gradient based SOC differs from surrogate optimization in that instead of directly acting as a surrogate model and making a direct prediction of the controlled variables, we instead make an estimate of the gradients and control based on those.

3 Implementation

The main steps for the development of the control structures are to simulate the heat exchanger network and use a trained Gaussian process to control the valve splits as the system is disturbed. The implementation can be separated into 3 parts, the plant and its data generation, the Gaussian Process and its process controller implementation, and the simulation where it all is connected.

3.1 The Plant

A Python implementation of the heat exchanger network was made using the CasADi package. This was used to simulate the plant and generate the information used to train and test the Gaussian process implementations. There were made some assumptions such as the use of Chen's approximation, to avoid the numerical issue of diving by zero when the temperature differences were the same. The approximation also avoids issues with negative temperature differences through the simulation.

Multiple datasets were generated with two different sample counts, 500 and 2500 samples. The code for the real model is shown in Appendix B.1, and the script to generate training and test datasets is shown in Appendix B.2. Note that different datasets were made for the unconstrained and constrained case.

3.2 Control structures

The implementation of the machine learning tool was created in Python using GPy [6], a framework designed to perform Gaussian process machine learning. The code for the Gaussian process is implemented in Appendix B.3. Two control structures were made for the unconstrained problem and three for the constrained problem. All structures share an overall design regime, and their names and symbols are shown in Table 3.1. The controller implementations are in Appendix B.4

Table 3.1: Control structures and their symbols.

Symbol	Name
u	Surrogate controller
g	Gradient controller
uc	Constrained surrogate controller
uc2	Constrained mixed controller
gc	Constrained gradient controller

The general workflow is that a controller takes a measurement, which is then pre-processed before given to the internal Gaussian process module which makes an prediction. Afterwards that prediction is post-processed into the new valve openings which are passed back to the simulation. Each controller is trained on a dataset at initialization and ready to be used in the simulation afterwards. [7]

For the unconstrained problem the two controllers are the surrogate controller (u) and the gradient control strucutre (g). Controller diagrams are shown in Figure 3.1 The surrogate controller takes in a measurement and directly predicts an optimal valve split. To reduce sensitivity to noise, the controller takes a strong weighted average between the new and old value, $u_{new} = 0.05 \cdot u_{old} + 0.95 \cdot u_{new}$. The gradient control structure takes a measurement and then predicts the gradients for system at that point. The gradients are controlled to zero with the use of an I-controller. The integral gain, k, was empirically tuned to -0.0005 for MS2 to MS4 and -0.001 for MS1.

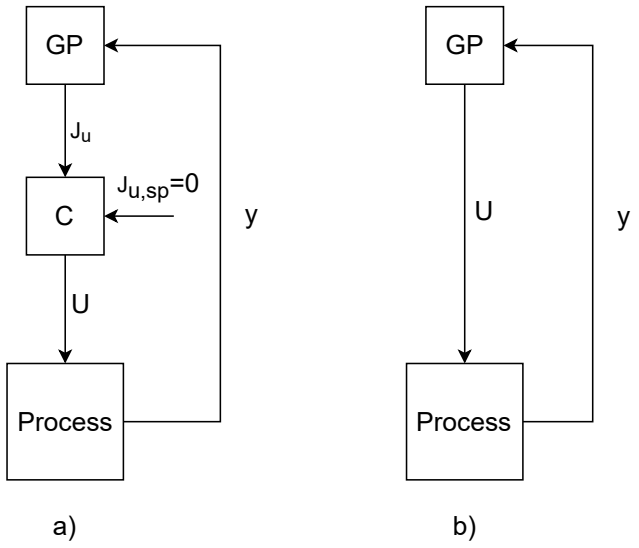


Figure 3.1: Illustration of: a) surrogate controller (u) the gradient control structure (g). GP is gaussian predictor and C is a setpoint controller. y is a measurement of the process.

The constrained control structures build upon those from the unconstrained case. However as there is a need to measure the temperatures of the streams out of each heat exchanger to make sure the constraints are met, the measurements used for the constrained case all contain the measurements T1, T2 and T3. This is further explained in Section 3.3.2.

The constrained gradient control structure (gc) uses the same gradient prediction as in the unconstrained case but implements an active constraint switching scheme depending on if the temperature constraint is active. The logic is shown in Figure 3.2. [8] [9] The active constraint switching was tuned empirically to achieve best performance based on control performance on plots of disturbance set 1.

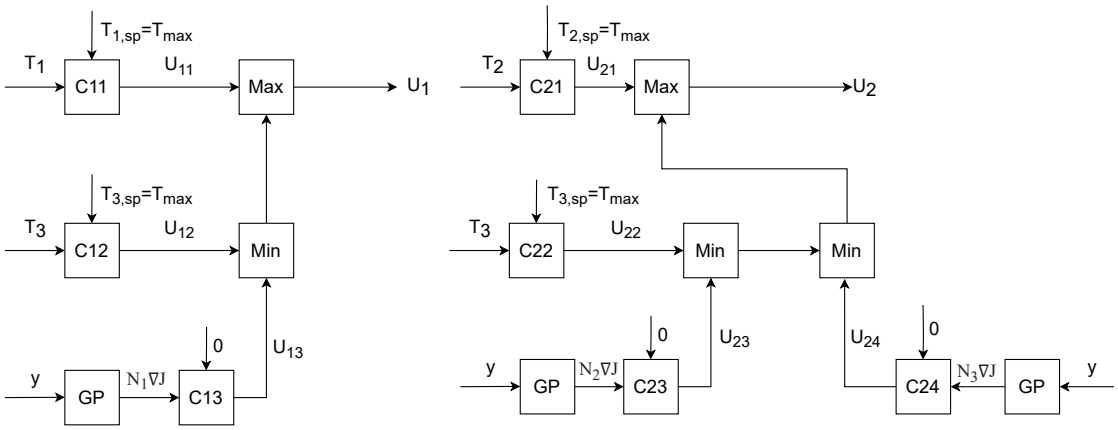


Figure 3.2: Illustration of selector logic of the constrained gradient (gc) control structure.

The constrained surrogate controller (uc) is similar to the unconstrained surrogate controller but was modified to train on a dataset which report constrained optimal valve openings. There were also implemented controller configurations which changes data processing of the measurements before they are passed to the GP module. These controller configurations also apply to the constrained mixed controller. The first configuration is the temperature configuration (configuration t), which simply passes the measurements to the Gaussian predictor. The second configuration is the temperature violation configuration (configuration tv), which replaces the measurements T1, T2 and T3 with constraint violations T1v, T2v, and T3v. These values are the difference between the temperature constraint and the measured temperature, which means that if the measured temperature is above the constraint, the temperature violation will be negative. The final configuration is the combined configuration ($full$ configuration), which is the same as the temperature constraint configuration but includes both the temperatures T1, T2, T3, and constraint violations T1v, T2v and T3v in the Gaussian prediction. The configurations are summarized in Table 3.2. Like the unconstrained case, the same weighted average of new and old valve opening was used to reduce noise sensitivity.

Table 3.2: Controller configurations, used with uc and uc2 controllers.

Symbol	Description
t	Temperature configuration
tv	Temperature violation configuration
$full$	Combined configuration, includes both temperatures and violations.

One important part of the configurations is that when the configuration is set to tv or $full$, the constrained surrogate controller (uc) will apply a back-off scheme which will dynamically add back-off to the constraint violation variables. If the prediction is still violating the constraint it will add extra back-off, and slowly release it when not violating the constraint. The implementation of the dynamic back-off is very basic, and

is shown in Appendix B.4. The mixed controller (uc2) does not use this dynamic back-off scheme for any of the controller configurations, but instead takes the active constraint switching approach from the gradient controller and re-purposes it to use on the valve predictions. The selector logic for the constrained mixed controller is shown in Figure 3.3.

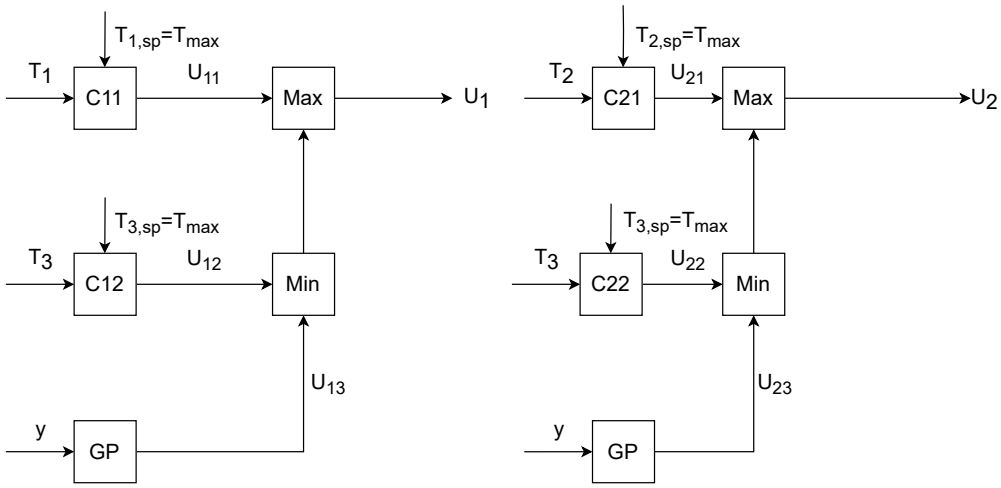


Figure 3.3: Illustration of selector logic of the constrained mixed controller (uc2) control structure.

3.3 Closed loop system simulation

At every time step in the simulation, the state of the system is calculated from the disturbances and the current valve openings. After resolving the system state, a set of measurements is taken, which are considered the real sensor data in a real world plant. These measurements are given to the current control structure which makes a prediction and gives back the adjusted valve openings. The real optimal valve splits are also calculated, and used to find the loss of the prediction over the course of the simulation. It is assumed that the dynamics of the system is negligible compared to that of the control structure, such that at every time step the system has reached steady state.

Noise can also be introduced to the measurements to simulate measurement error in a real plant. Two cases are considered, one without any noise and one where noise is introduced both during training and on the measurement structures in the simulation. The noise applied was gaussian with a range of ± 1 in all cases.

3.3.1 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:

$$\mathrm{J} = -\mathrm{T} \tag{3.1}$$

where J is the cost.

In the context of machine learning models, the term loss is usually defined as the metric which measures the fit of these models to the data, and in traditional ANNs this loss is used to update the weights that decide the output. The loss would be the objective one would try to minimize or maximize through the training process. In this work, however, we use the term loss referring to the optimality of the current operation of a process. For an unconstrained case, the normal loss is simply defined as the difference between the predicted and optimal cost.

$$\text{Loss} = \text{J} - \text{J}^* \tag{3.2}$$

Where J^* is the optimal cost found from the accurate model. However, when working with the constrained case there is the potential for the loss to exceed the optimal temperature by violating the constraints imposed. As such the loss is split into two parts, the normal loss (Equation 3.3) and the constrained loss. (Equation 3.4)

$$\text{Loss} = \text{J} - \text{J}^* \qquad \text{if J}^* <= \text{J else 0} \tag{3.3}$$

$$\text{Constrained Loss} = \text{J}^* - \text{J} \qquad \text{if J}^* > \text{J else 0} \tag{3.4}$$

As the simulation happens over time, the respective losses were integrated over time using the trapezoidal rule shown in Equation 3.5, where the Δt is the time step and i the iteration step, and Loss_i the loss at that time iteration.

$$\text{Loss (integrated)} = \sum_{i=1}^{n} \frac{\text{Loss}_i + \text{Loss}_{i-1}}{2} \Delta t \tag{3.5}$$

3.3.2 Measurement sets

Four measurement sets were used, to see how well each set of measures helped the Gaussian process make predictions, however for the constrained case the measurement sets were modified to include T_1, T_2 and T_3. The most accurate measurements are temperatures, but measurement set 3 included heat capacity to see how well performance was when the heat capacity of the streams were known, however this measurement can be inaccurate in practice. The measurements sets in Table 3.3 show what measurements each set contains. Negative measurement sets are for the constrained case.

Table 3.3: Table showing which measurements that are in each measurement table. Negative measurement sets are used for the constrained case.

MS	Measurements
1	T_0, T_1, T_2, T_3, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$
2	T_0, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$, $T_{he,1}$, $T_{he,2}$, $T_{he,3}$
3	T_0, T, $T_{he,1}$, $T_{he,2}$, $T_{he,3}$, w_0, w_1, w_2, w_3
4	T_0, T, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$, α, β
-1	T_0, T_1, T_2, T_3, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$
-2	T_0, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$, $T_{he,1}$, $T_{he,2}$, $T_{he,3}$, T_1, T_2, T_3
-3	T_0, T, T_1, T_2, T_3, $T_{he,1}$, $T_{he,2}$, $T_{he,3}$, w_0, w_1, w_2, w_3
-4	T_0, T, T_1, T_2, T_3, $T_{h,1}$, $T_{h,2}$, $T_{h,3}$, α, β

Optimization that run close to the optimal should be possible through ordinary optimizations methods such as controlling a cost gradient, with just temperature measurements.[10] Along with that, temperatures are easy to measure and would save a lot effort on the measurement side of implementing a control system. Thus 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)

3.3.3 Disturbance sets

Three disturbance sets were made to test the process controllers. Disturbance set 1 is the benchmark set which cycle the individual disturbances to their largest deviations from the nominal state, individually, to measure how well the controllers responds to each disturbance. Two versions of disturbance set 2 were made. Each version of disturbance set 2 picks disturbance points from a test dataset from the unconstrained or constrained case. Using the dataset, some sets of disturbances are selected and interpolated between over time, simulating operation with multiple active disturbances that change the simultaneously. The implementation of each disturbance set is in Appendix B.6. The disturbance set 2 used for unconstrained case contained disturbance combinations which could lead to infeasible operation for the constrained case, thus a dataset with disturbances which were known to be within the feasible range was used for the constrained case.

3.4 Testing

The simulation was set up with the following variations: measurement sets, disturbance sets, noise cases, training sample sizes, and controller types with their configurations.

First the unconstrained case was tested using disturbance set 1. The effects of having optimal data in the training sets for the surrogate controller (u) were investigated. Afterwards the significance of number of training samples was investigated as well as effects of measurement noise on the predictions. Then each controller and measurement combination was tested to find out what disturbances are better handled by the control structures. Then, disturbance set 2 was used to measure a more realistic "real world" performance. From those results the best controller and measurement set combinations were selected.

In the constrained case, disturbance set 1 was used to investigate sample size and its effects on the constrained system before the controller configurations were compared. Then, similarly to the unconstrained case, disturbance set 2 was used with the findings to analyse and select the best performing controllers, configurations and measurement set combinations. Lastly, the effect of noise on the best controller and measurement combinations were investigated.

4 Results

4.1 Case: Unconstrained

4.1.1 Optimal data distribution

In the case of unconstrained surrogate optimization using the surrogate controller (u), the training data had a percentage of its training points shifted close to the optimal operating point by changing the valve openings closer to the optimum. This was made on the assumption that having known measurements near the optimal operating point would give better predictions in the areas that mattered. On the other hand, it was also assumed that there was a need to have some points further away from the optimum to ensure predictions converged properly to the optimal operating point. Four cases were tested, one with measurements from optimal data, one with only random measurements, and two cases with a mix of 30% and 70% of the measurements close to the optimum. The first run was with 500 training samples, on disturbance set 1.

From the Table 4.1 we can see that MS1 remained somewhat unchanged, with the exception of the divergence with only optimal data. There was no clear trend in the data, but data distributions which at least contained some optimal data had the better performance. The advantages vary depending on the measurement set. The use of only optimal data gave good results for MS2 and MS3, but the advantage was not as large for MS3. For MS2 the best results were obtained with 70% optimal measurements. Measurement set 4 had the largest impact with respect to the training data distribution, presenting large errors when using only optimal data, and performing decently with mostly random data.

Table 4.1: Loss for simulations, surrogate controller (u) trained on 500 samples with different distributions of measurements at the optimal and random operating points. Measurement noise was applied. Note that divergence happened for MS1 with only optimal data.

MS	Random	30 % optimal	70% optimal	Only optimal
1	18.46	18.34	18.67	-
2	19.68	15.97	14.16	15.11
3	18.32	15.83	16.42	15.28
4	22.60	18.85	34.28	732.07

The suggestion is therefore to have some optimal data in the training dataset. If one increases the number of training samples, the distribution will matter less and likely not react as strongly to the data distribution, since the chances for important datapoints for training do not get left at the wrong position. Therefore the number of training samples were increased to 2500 samples, and the experiment rerun for the different training configurations. The configuration with only optimal data was skipped as MS1 still diverged.

Table 4.2: Loss for simulations, surrogate controller (u) trained on 2500 samples with different distributions of measurements at the optimal and random operating points. Measurement noise was applied.

MS	Random	30 % optimal	70% optimal
1	18.33	18.44	19.54
2	22.36	14.11	13.62
3	21.50	16.86	16.52
4	19.00	20.96	82.30

With the larger training sample count, some trends could be observed from each measurement set. For MS1 and MS4, the error increased with more optimal data, while the opposite was seen for MS2 and MS3. Going forward, the configuration with 30% optimal data were be used for further testing on the unconstrained dataset, as it yielded the most balanced performance across the measurement sets. It can however be noted that the 70% optimal data configuration yielded best results for MS2 and MS3.

4.1.2 Training size and controllers

A comparison was done to determine if 500 training samples were sufficient, or if a substantial gain was to be had from an increased number of samples. Training with and without noise on measurements were tested. The data distribution for training at 30% optimal values was used for the surrogate controller.

Table 4.3: Loss per controller on the unconstrained optimization on disturbance set 1. No noise in the system. Two sample sizes were used for training the controllers. Surrogate controller trained on 30% optimal measurements.

MS	Controller	500 Samples	2500 Samples
1	u	17.85	18.14
	g	17.15	16.82
2	u	15.43	13.50
	g	55.80	40.42
3	u	13.08	13.03
	g	61.79	48.06
4	u	18.49	19.92
	g	46.04	32.00

For the noise free case, results are shown in Table 4.3. The general trend is that the gradient control structure (g) performed worse than the surrogate controller (u). This was likely because there is not a need to converge to values, when the surrogate controller can predict the optimal split configuration with a single iteration. So,

unlike the gradient control structure, the surrogate controller will not spend much time converging towards the optimal state, reducing loss. Given that this is disturbance set 1, the disturbances spike hard, which is much less favorable for the gradient control structure. Disturbance set 2 which features more gradual changes over time may show more realistic real world operation for the gradient based solution.

MS1 did not improve performance with more samples when using the surrogate controller, showing a small increase in loss instead, but gain was seen on the gradient control strcutures. MS4 also saw some increase in loss with the surrogate controller. This could be due to the randomness in the measurements or that the training points were different. For the rest of the measurement sets the predictions improved with a larger training dataset. Additionally, seen in Table A.1 the same simulations were conducted with 70% optimal measurements for the surrogate controller, which resulted in small improvements in MS2 and MS3, but a drop in performance for MS4.

Table 4.4: Loss per controller on the unconstrained optimization on disturbance set 1. Training and measurements had noise applied. Two sample sizes were used for training the controllers. Surrogate controller trained on 30% optimal measurements.

MS	Controller	500 Samples	2500 Samples
1	u	18.11	18.44
	g	23.08	17.99
2	u	15.74	14.11
	g	54.03	42.62
3	u	15.14	16.86
	g	54.01	44.62
4	u	18.70	20.96
	g	49.28	31.41

When measurement noise was introduced on training samples and on the measurements during simulations, seen in Table 4.4, there was not a significant change in performance from the unconstrained case. Overall the changes were not large, but in general a small drop in performance was observed. Particularly MS3 and MS4 saw some degradation with more samples using the surrogate controller (u).

4.1.3 Disturbance sensitivity

By plotting the closed-loop system response to disturbance set 1, and how the predictions of the controller behave compared to the actual optimum, it was possible to observe which disturbances that each controller and measurement set was sensitive to. For visual clarity noise free simulations were used. Given the similar performance, 500 datapoints were used for the simulations. First the surrogate controller (u) was tested.

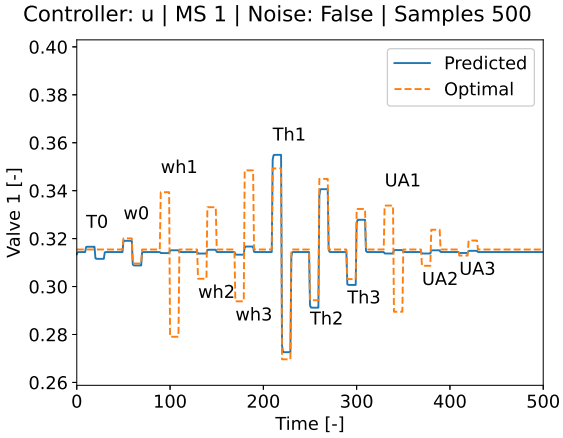


Figure 4.1: MS1: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

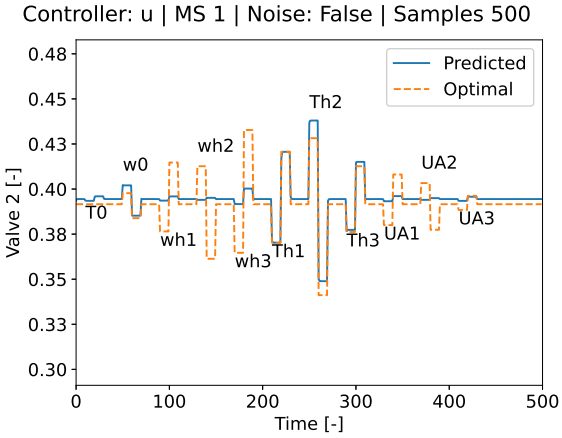


Figure 4.2: MS1: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS1 are shown in Figure 4.1 and 4.2. The surrogate controller reacted well to changes in the input heat capacity (w0). It responded weakly to changes in the heat exchangers overall heat transfer coefficients. It seemed to respond to changes in the input stream temperature, even though the optimal valve openings do not. Heat capacity of the hot streams was not handled well. As seen in Figure 4.4 the loss increases the most when there are changes in the heat capacities, which is the largest weakness of this measurement set. The output temperature is shown in Figure 4.3.

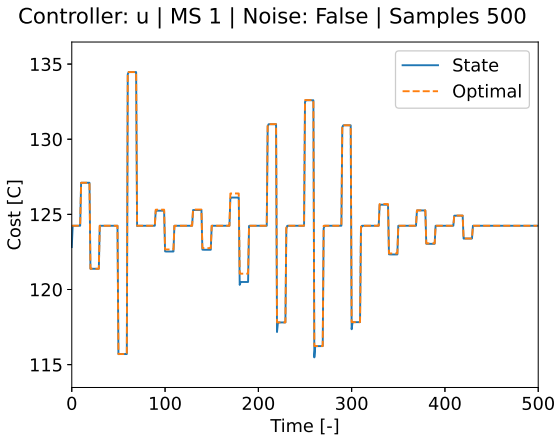


Figure 4.3: MS1: Plot of output temperature compared to the optimal temperature.

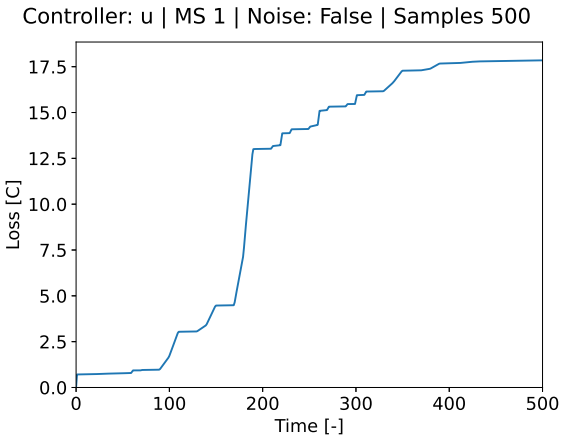


Figure 4.4: MS1: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

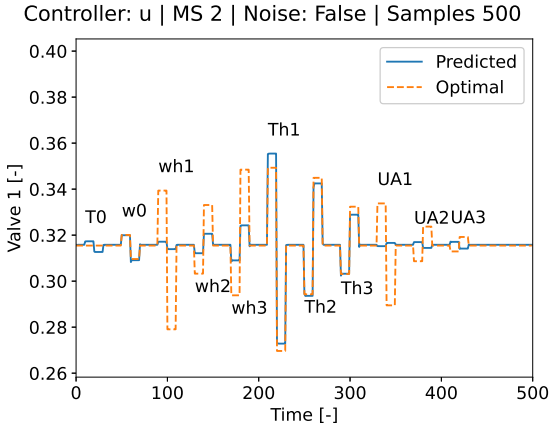


Figure 4.5: MS2: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

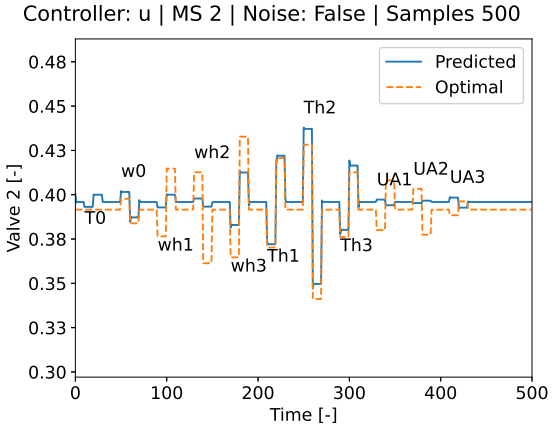


Figure 4.6: MS2: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS2 are shown in Figure 4.5 and 4.6. The disturbance sensitivity was similar to MS1, though here the controller responded in the opposite direction when changes in the overall heat transfer coefficients occurred. Still, the largest loss was incurred with the changes heat capacity as seen in Figure 4.8.

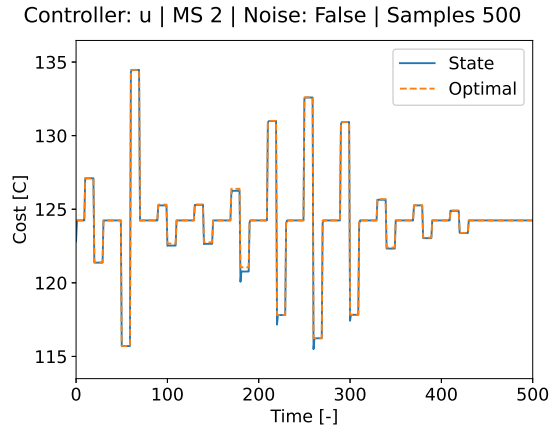


Figure 4.7: MS2: Plot of output temperature compared to the optimal temperature.

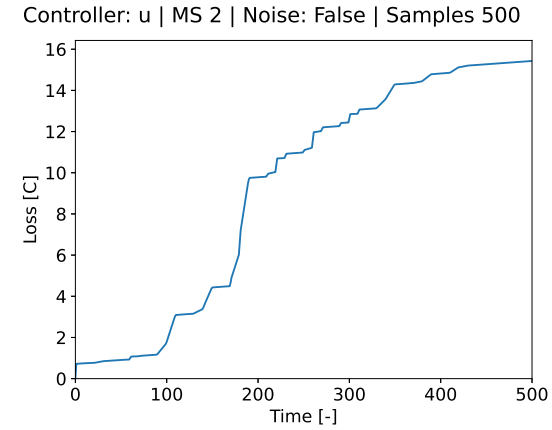


Figure 4.8: MS2: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

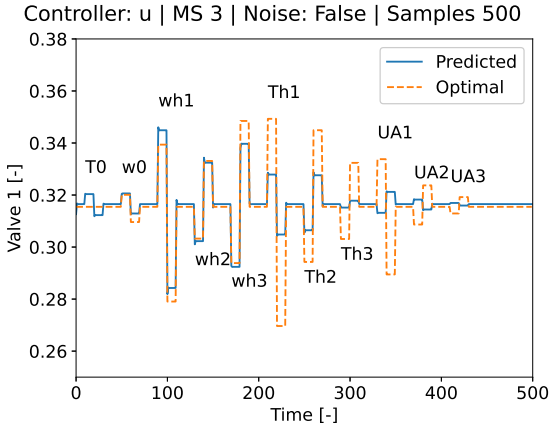


Figure 4.9: MS3: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

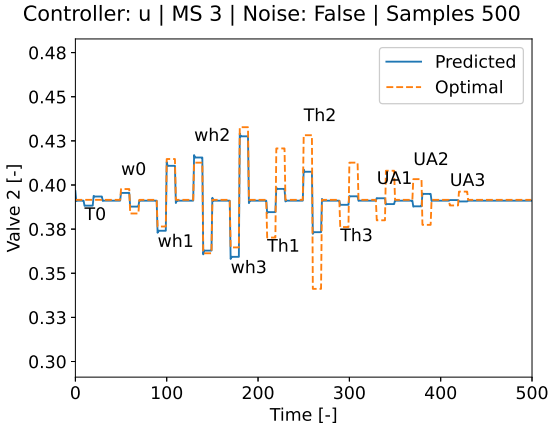


Figure 4.10: MS3: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS3 are shown in Figure 4.9 and 4.10. Since MS3 includes the heat capacity, the resulting control structure had much less trouble controlling for those disturbances, but instead performed less favorably with changes in the temperature in the hot streams into each heat exchanger. The changes in temperature for the hot streams were the largest contributor of loss in Figure 4.12. The controller action was similar to MS2, with the opposite reaction to a change in a heat exchangers overall heat transfer coefficient.

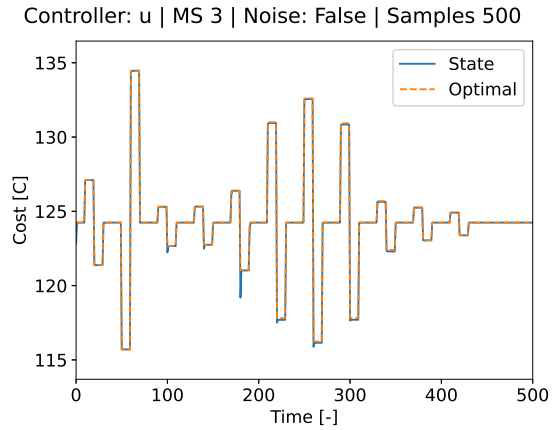


Figure 4.11: MS3: Plot of output temperature compared to the optimal temperature.

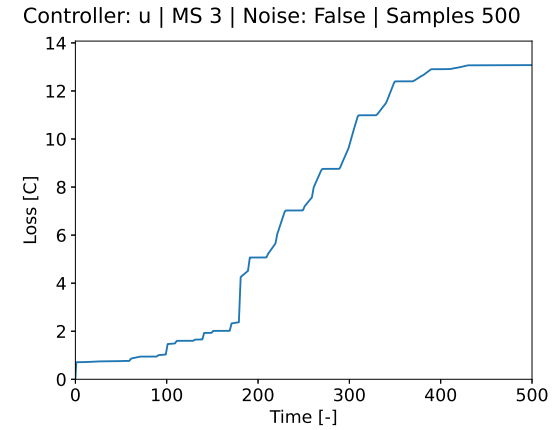


Figure 4.12: MS3: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

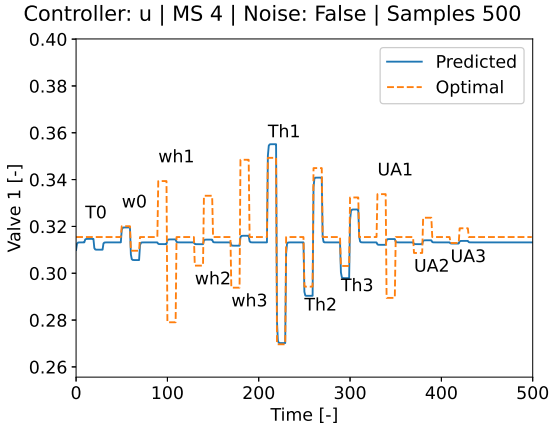


Figure 4.13: MS4: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

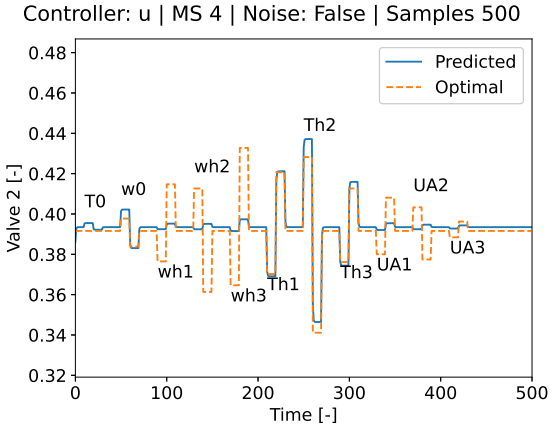


Figure 4.14: MS4: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS4 are shown in Figure 4.13 and 4.14. Like MS1 and MS2, the change in heat capacity caused the biggest loss, especially with changes in heat capacity for the third heat exchanger, as seen in Table 4.16. Other than that, the surrogate controller reacted weakly in the presence of overall heat transfer coefficient changes.

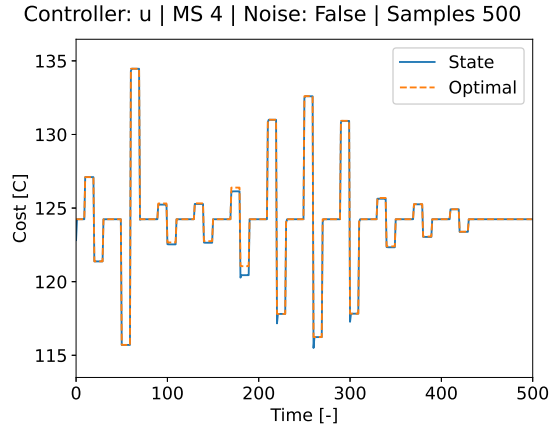


Figure 4.15: MS4: Plot of output temperature compared to the optimal temperature.

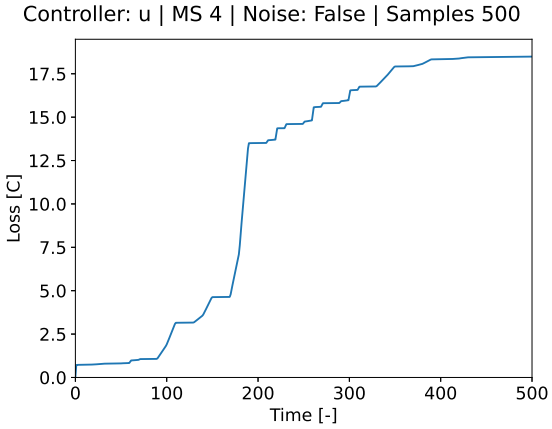


Figure 4.16: MS4: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

For the gradient control structure, the controlled variable was the gradient itself. Gradient 1 is the gradient calculated with relation to the first input and gradient 2 is the gradient calculated with relation to the second input. The gradients for MS1 are shown in Figure 4.17 and 4.18. The gradients are not scaled, and the simulation was done without noise for simplicity. The actual gradient is relatively low, which indicated they are close to the optimum, but it also means the point the control structure converged to is not the true optimum. However, gradient plots do not show performance well, so as with the surrogate controller, the updated valve splits have been used to show disturbance sensitivity and general performance. The gradients for the remaining simulations are shown in Appendix A.2

The scaled gradients are used to update the respective valve openings. In disturbance set 1, the change in each disturbance were relatively large and instant, which made convergence slower for the gradient based controller. However, faster updates would not be needed under operation where disturbances do not change as rapidly, such as with disturbance set 2. For simplicity, only 500 samples were used, but as seen in Table 4.3, an increase in samples could show improvements in performance of the gradient controller.

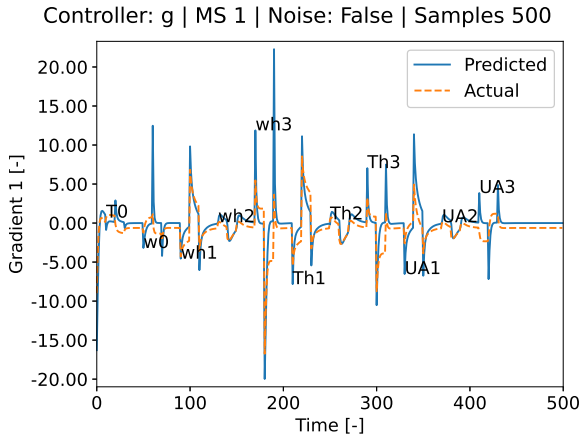


Figure 4.17: MS1: (Gradient 1) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

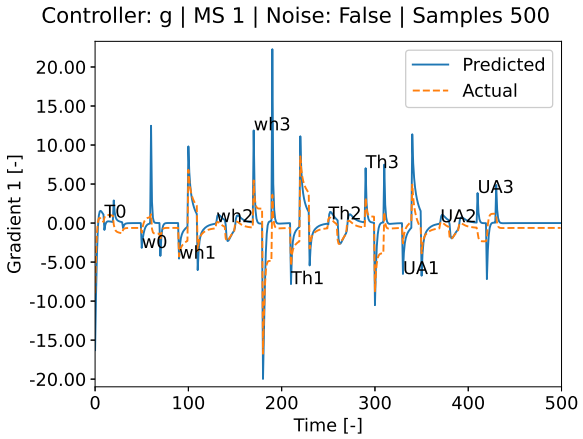


Figure 4.18: MS1: (Gradient 2) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS1 are shown in Figure 4.19 and 4.20. Like seen during evaluation of the training dataset size in Table 4.4, the gradient control structure (g) only outperformed the surrogate controller (u) with MS1. Based on both the valve positions here, and the gradients in Figures 4.17 and 4.18, the gradient controller is able to accurately react to most of the disturbances. The increase in input temperature did not get handled properly, but unlike the surrogate controller (u) there was generally some reaction to a change in all other disturbances. The heat capacity was better handled, and so was the changes in the overall heat transfer for each heat exchanger.

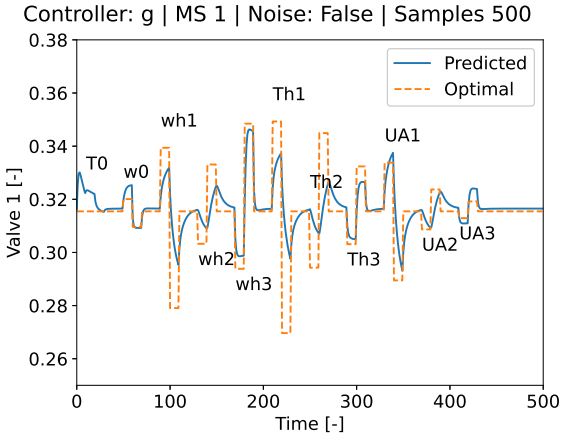


Figure 4.19: MS1: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

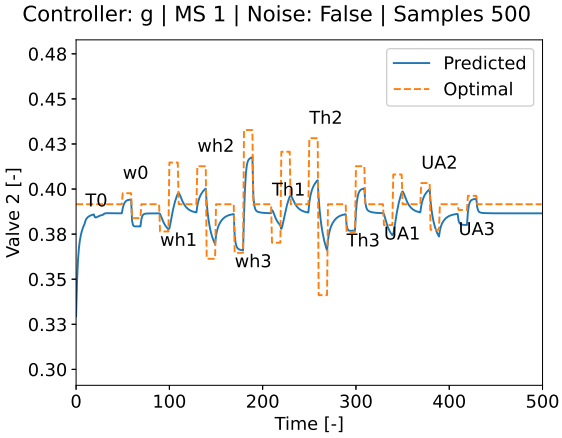


Figure 4.20: MS1: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

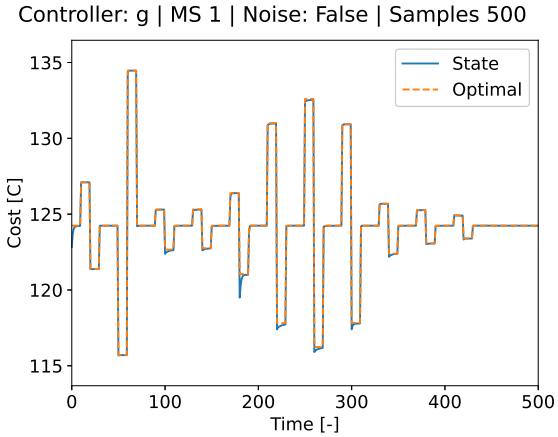


Figure 4.21: MS1: Plot of output temperature compared to the optimal temperature.

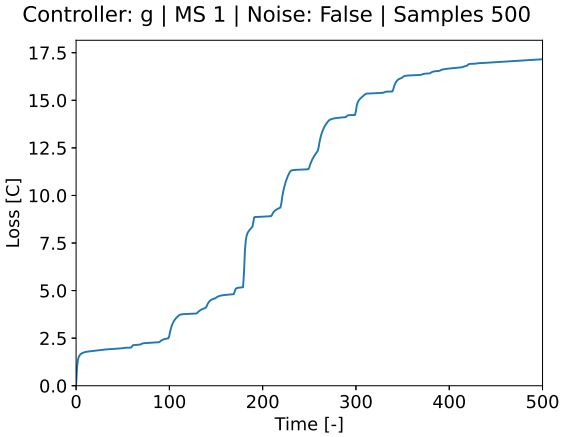


Figure 4.22: MS1: Integral loss of simulation for gradient controller. Regions with high rise in loss are where bad predictions cause considerable deviations from optimum.

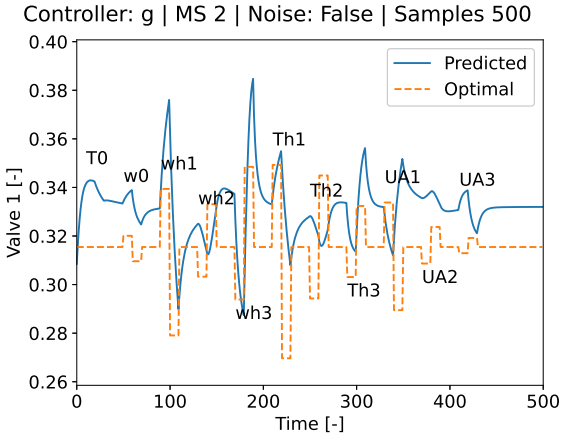


Figure 4.23: MS2: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

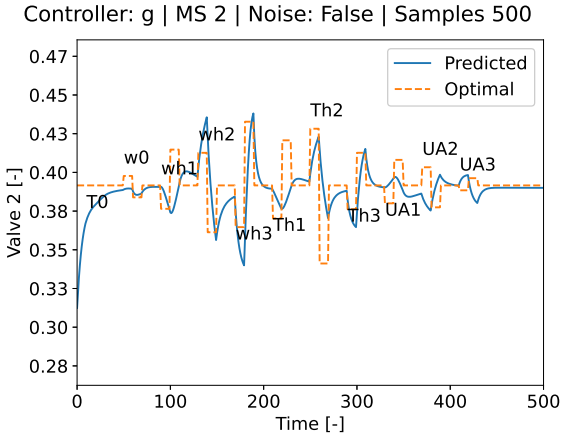


Figure 4.24: MS2: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS2 are shown in Figure 4.23 and 4.24. MS2 did not behave as well as MS1, where especially the nominal state valve position had a big offset for the first valve. It did react similarly to MS1, by showing changes in valve positions at most disturbances, but it was far less clear if those changes were good changes, given the big offset with relation to the optimal values. Based on Table 4.4, the gradient controller did not perform well for MS2 to MS4. Noticeable drops in optimal temperature can be seen in Figure 4.7, and the loss increased steadily over the course of the simulation as seen in Figure 4.26.

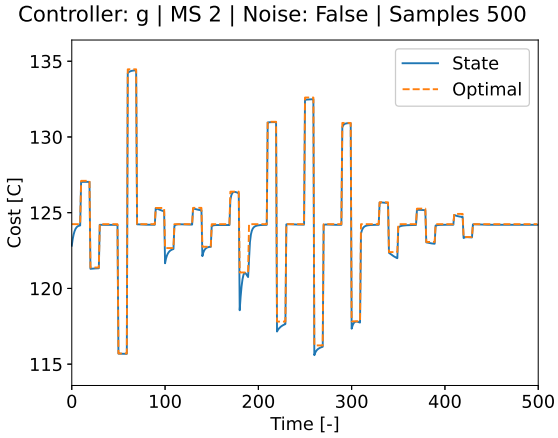


Figure 4.25: MS2: Plot of output temperature compared to the optimal temperature.

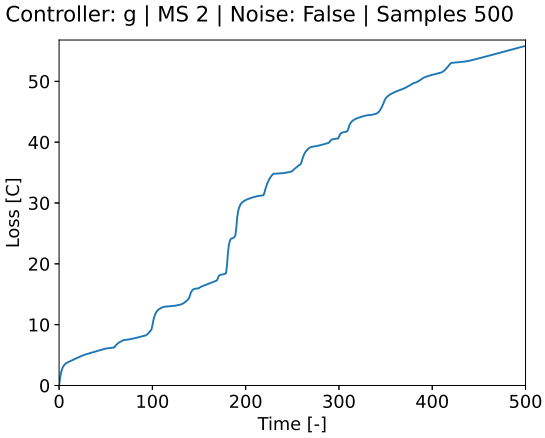


Figure 4.26: MS2: Integral loss of simulation for gradient controller. Regions with high rise in loss are where bad predictions cause considerable deviations from optimum.

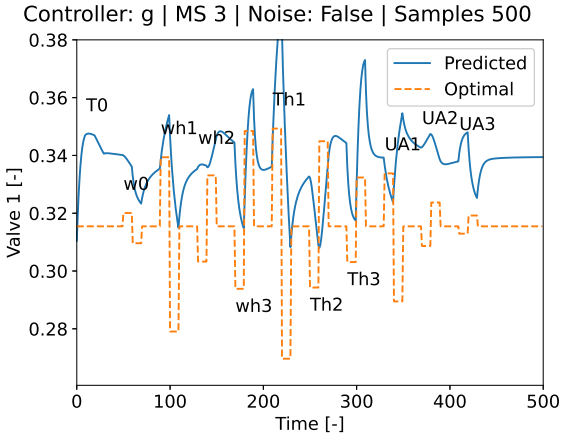


Figure 4.27: MS3: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

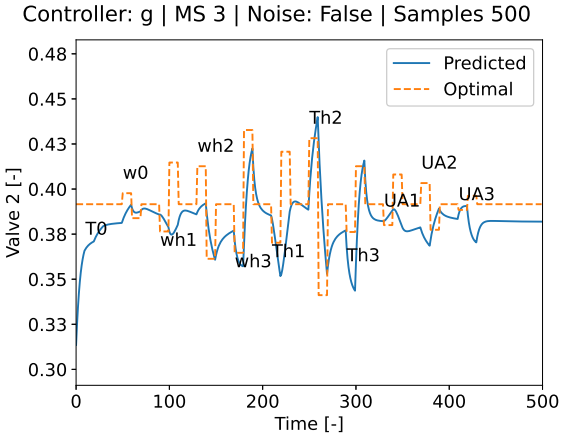


Figure 4.28: MS3: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS3 are shown in Figure 4.27 and 4.28. The gradient controller performed worse for MS3 than MS2 in terms of loss. The offset at the nominal state was bad for both valves.

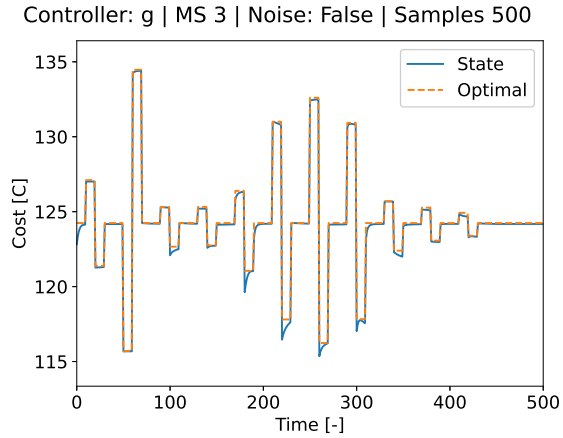


Figure 4.29: MS3: Plot of output temperature compared to the optimal temperature.

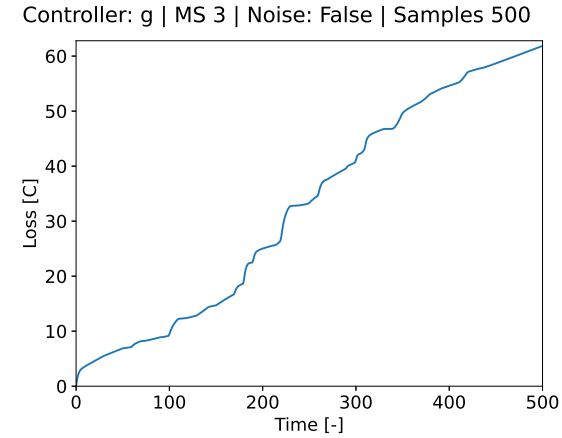


Figure 4.30: MS3: Integral loss of simulation for gradient controller. Regions with high rise in loss are where bad predictions cause considerable deviations from optimum.

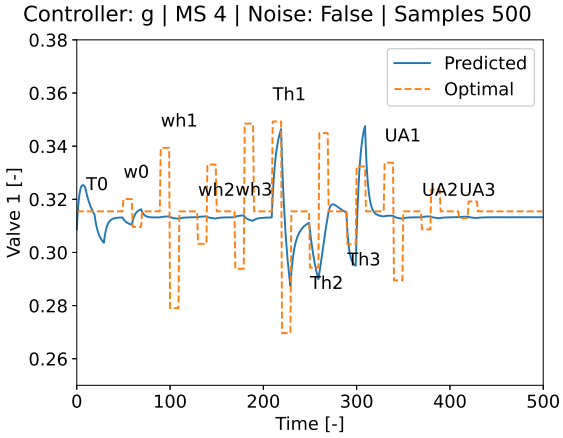


Figure 4.31: MS4: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

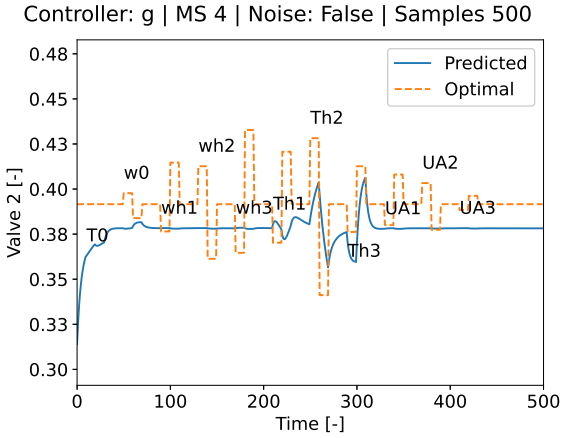


Figure 4.32: MS4: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

The valve positions for MS4 are shown in Figure 4.31 and 4.32. MS4 was also not performing well, and while it reacted to hot stream temperature changes, it was insensitive to heat capacity and overall heat transfer changes. MS4 also showed bad performance for the nominal state for valve 2.

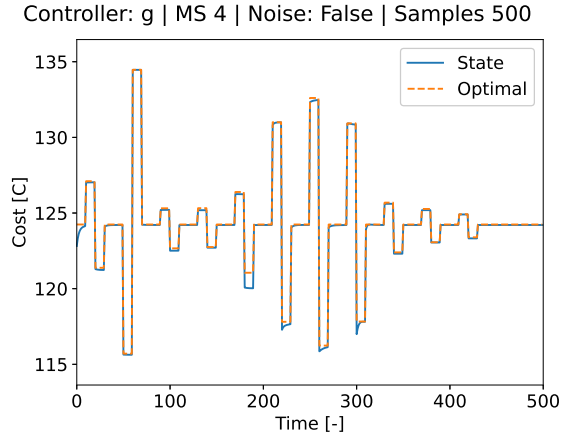


Figure 4.33: MS4: Plot of output temperature compared to the optimal temperature.

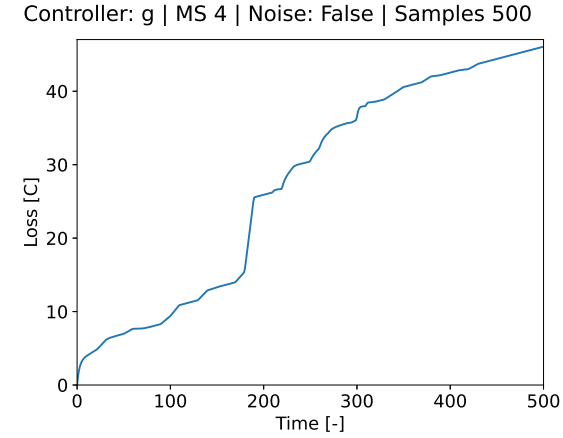


Figure 4.34: MS4: Integral loss of simulation for gradient controller. Regions with high rise in loss are where bad predictions cause considerable deviations from optimum.

4.1.4 Disturbance set 2

Disturbance set 2 shifts gradually over time between disturbances in a test dataset, and simulates a more realistic operation with multiple disturbances active at the same time. Like with disturbance set 1, simulations were run with different training sample sizes. Unlike disturbance set 1, disturbance set 2 used test points which have disturbance deviations that could be up to 20% larger than those in the training dataset. The same configuration for the surrogate controller was used, at 30% optimal data. The test was performed with noise enabled, but results were not significantly different from the noise free case. Data for noise free case can be seen in Appendix A.2.

The integral gain for gradient controller was set to -0.001 due to a significant increase in performance for MS1, the other measurement set constants were kept at a value of -0.0005. The performance result for the other controllers using -0.001 as integral grain is shown in Appendix A.3.

Table 4.5: Loss per controller on the unconstrained optimization on disturbance set 2. Noise was applied to training and measurements. Two sample sizes were used for training the controllers. Surrogate controller trained on 30% optimal measurements.

MS	Controller	500 Samples	2500 Samples
1	u	119.32	116.67
	g	57.03	51.62
2	u	81.53	69.71
	g	290.22	260.74
3	u	70.69	60.53
	g	317.69	294.21
4	u	122.36	119.89
	g	230.74	174.18

The results for each controller structure is shown in Table 4.5. The overall loss did not change significantly for the surrogate controller as the training dataset increased to 2500 samples, but the drop was noticeable for MS2 and MS3 which scored the lowest losses of all configurations.

As seen from the disturbance sensitivity analysis, the gradient controller had a good response to most disturbances (Figure 4.19) and outperformed the surrogate controller with MS1. It was also the third lowest loss overall for the unconstrained problem.

In order to make the section more concise, only the most relevant plots have been included showing the performance of the best performing controllers for each measurement set. For practical reasons, the plots were generated with 500 training samples and no noise to show performance.

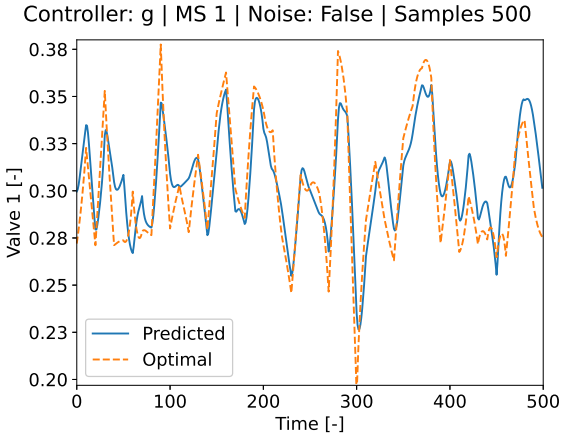


Figure 4.35: MS1: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

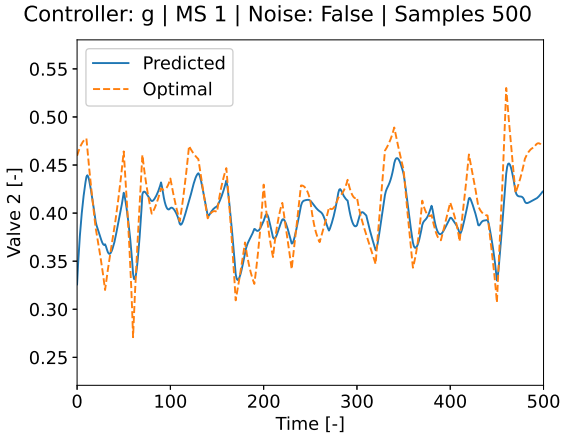


Figure 4.36: MS1: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

As can be seen in Figures 4.35 and 4.36, the controlled valves do not completely adjust to the optimum, as more disturbances are off their normal state, making it harder to predict disturbances and thus leaving more prediction errors. There was not a noticeable drop in performance on the output temperature in Figure 4.37, but the the integral of the loss rose steadily throughout the simulation as seen in Figure 4.38. Compared to disturbance set 1, the overall loss was higher.

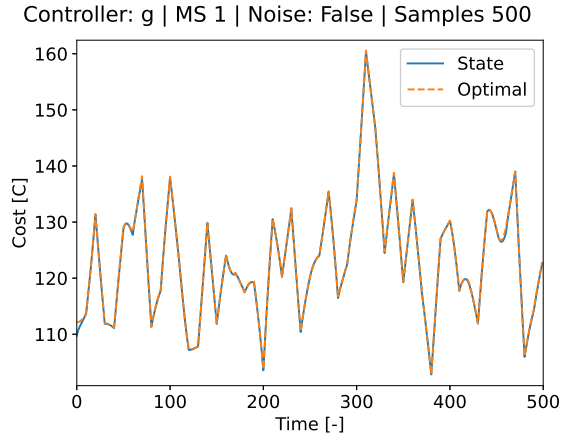


Figure 4.37: MS1: Plot of output temperature compared to the optimal temperature.

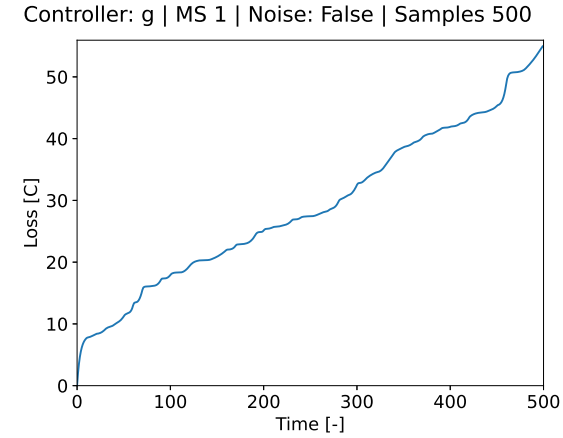


Figure 4.38: MS1: Integral loss of simulation for gradient controller. Regions with high rise in loss are where bad predictions cause considerable loss.

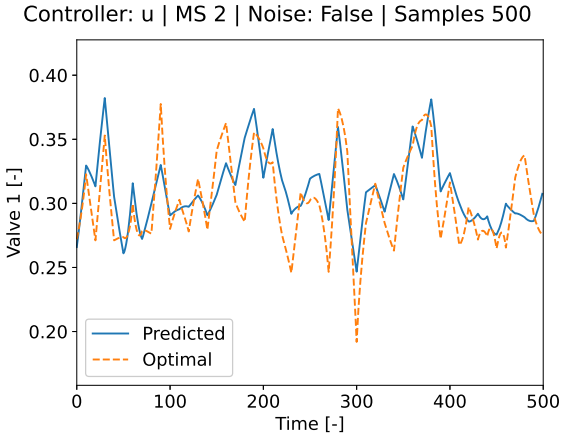


Figure 4.39: MS2: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

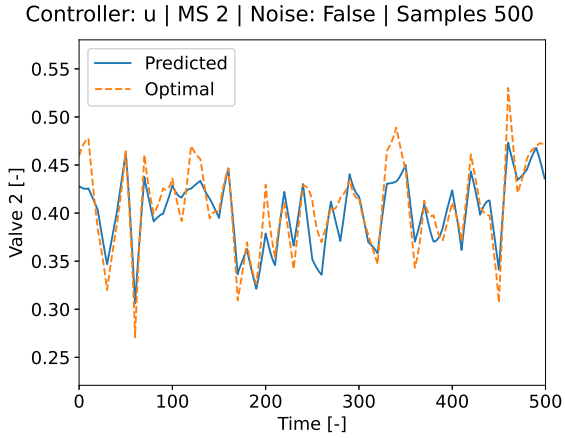


Figure 4.40: MS2: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

As can be seen in Figures 4.39 and 4.40, the surrogate controller (u) working with MS2 did not perform as well as the gradient controller (g) did on MS1. The controller makes reasonable predictions, but some points presented an inverse response with relation to the direction of the optimum. The predictions for the second valve followed the trend of the optimum for most of the simulation.

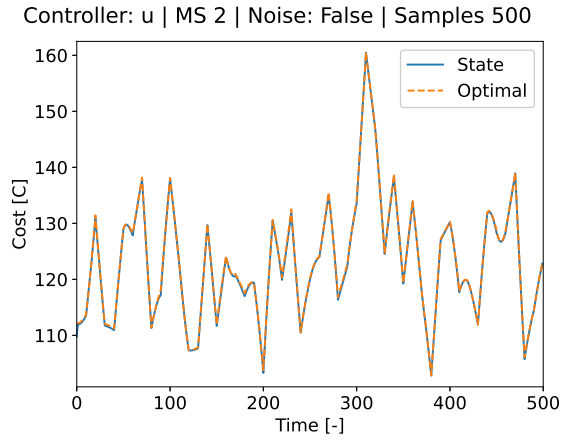


Figure 4.41: MS2: Plot of output temperature compared to the optimal temperature.

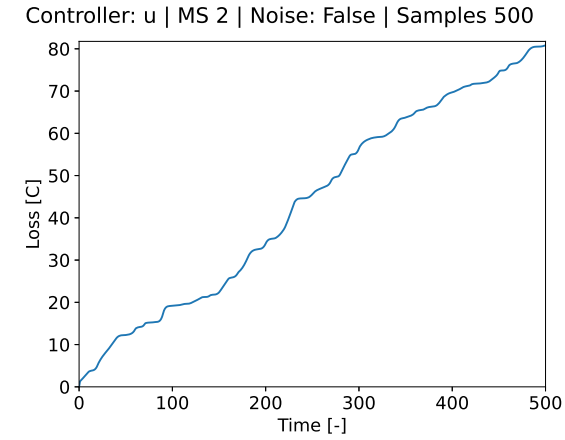


Figure 4.42: MS2: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

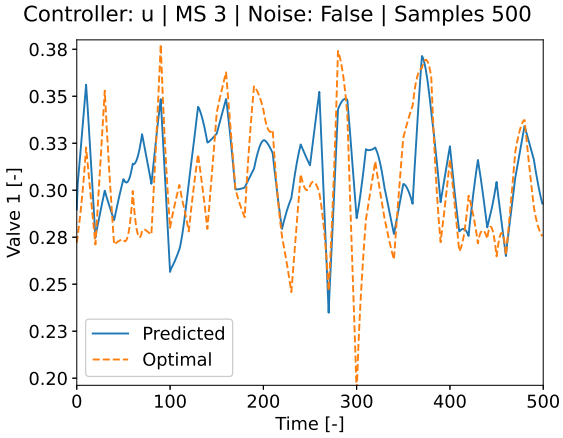


Figure 4.43: MS3: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

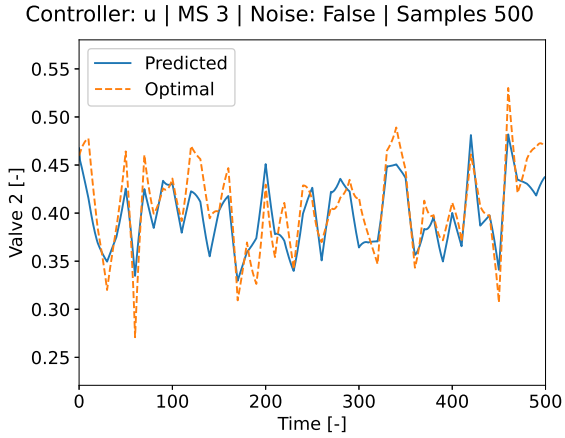


Figure 4.44: MS3: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

Figures 4.43 and 4.44 show the performance of the surrogate controller (u) subject to MS3 with disturbance set 2. Similarly to MS2, the predicted value followed the trend of the optimal valve, but showing prediction errors.

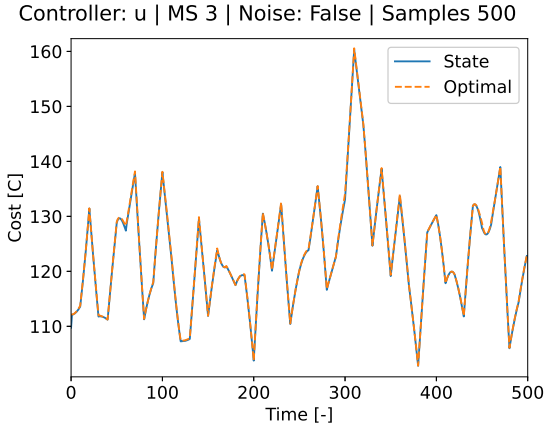


Figure 4.45: MS3: Plot of output temperature compared to the optimal temperature.

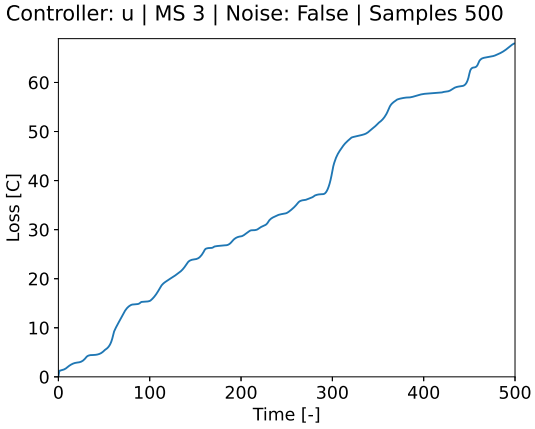


Figure 4.46: MS3: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

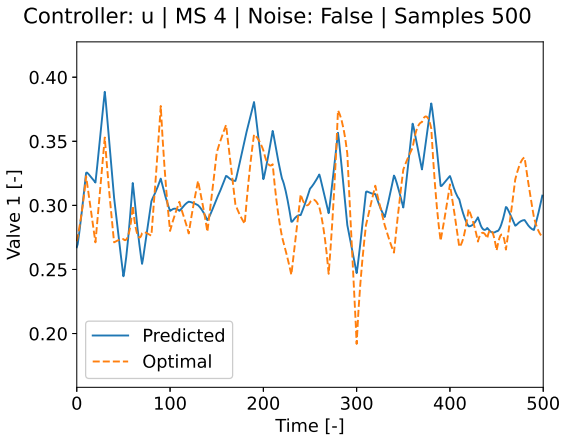


Figure 4.47: MS4: (Valve 1) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

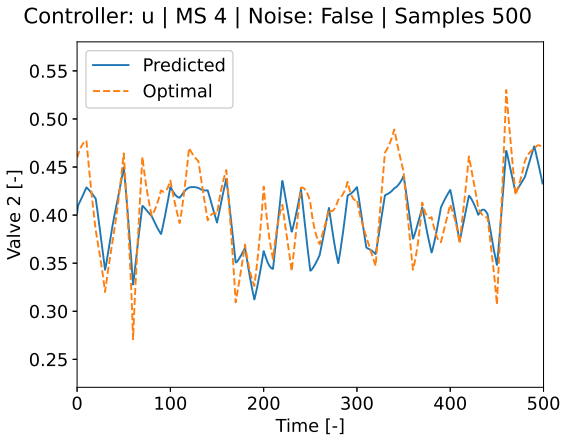


Figure 4.48: MS4: (Valve 2) Cycling through disturbances as they change. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

Figure 4.47 and 4.48 show the performance of the surrogate controller on MS4 with disturbance set 2. The total loss was overall larger than observed with the other measurement sets.

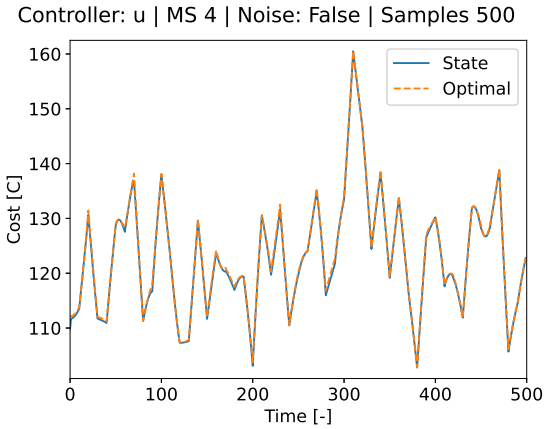


Figure 4.49: MS4: Plot of output temperature compared to the optimal temperature.

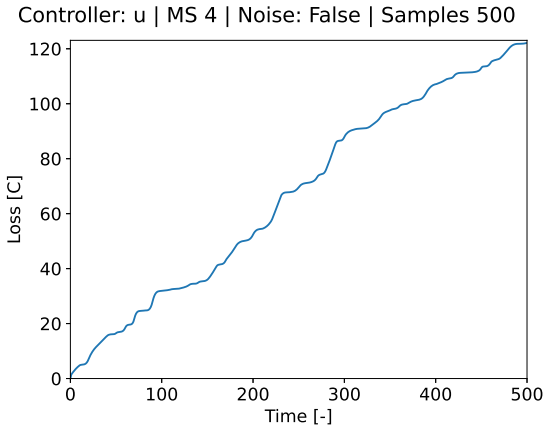


Figure 4.50: MS4: Integral loss of simulation for surrogate controller. Regions with high rise in loss are where bad predictions cause considerable loss.

Additionally, given that the surrogate controller showed better performance for MS2 and MS3 with disturbance set 1 when trained on more optimal data, a new run on disturbances set 2 was done. This time the surrogate controller was trained with 2500 datapoints, with 70% being optimal data compared to 30% like Table 4.5.

Table 4.6: Loss for surrogate controller trained with 2500 samples. 30% and 70% optimal data configurations, using MS2 and MS3. Disturbance set 2 was used. Noise was applied.

MS	Controller	2500 Samples
2	u (30%)	69.71
2	u (70%)	53.41
3	u (30%)	60.53
3	u (70%)	53.87

As can be seen, with more optimal data the performance of both MS2 and MS3 increased, and the performance between the measurement sets was nearly the same with when using 70% optimal data for training.

4.2 Case: Constrained

For this set of results, noise was enabled, including in plots, to further investigate real world performance in the presence of that noise and how well the temperature constraint could be met when this noise was affecting the predictions.

4.2.1 Training size

The surrogate controllers (u) optimal data split is assumed to be the same for the constrained case. Working within the constrained case, the training data for the surrogate controller (uc) were adjusted to consider the constrained maximum output temperature of each heat exchanger. This was necessary to ensure that the controller learned the constrained rules. The constrained gradient control structure (gc) did not use a modified training dataset, as it used a selector structure to account for the temperature limit. The constrained mixed controller (uc2) which applied active constraint selection used the same training data as the constrained surrogate controller (uc). The introduced constraint loss is considered the area above the constrained maximum temperature on the stream out of the network, and is here considered to be a measure of overall constraint violation. Additionally, the heated streams out of each heat exchanger is plotted and shown along with the temperature limit to see if the constraint is met.

Unlike the unconstrained problem, having the lowest normal loss is not enough to quantify performance of the predictions, as this loss would be zero when the constraint is violated and the output temperature ends up being larger than what would be possible with the constraint. The constraint loss is therefore a metric of the constraint violation, which also needs to be as low as possible.

To investigate the best performance of the constrained dataset, the number of training samples was revisited. In this configuration, controllers used negative measurement sets, where the three temperatures out of each heat exchanger were included. In Table 4.7 it can be seen that an increase in samples tend to help with the regular loss and with the constraint loss. Particularly MS-1 showed a decrease in constraint loss, as well as the normal loss. The constrained surrogate (uc) and mixed (uc2) control structures benefited most from the larger samples size, while the constrained gradient control structure (gc) did see a very small decrease using MS-3 and MS-4.

To qualitatively assess the performance gained with the increase of samples, the surrogate controllers (uc) stream temperatures have been plotted for the two sample counts, using MS-1. From the Figures 4.51 and 4.52 it can be seen that the controller trained with more samples is more capable of keeping the temperature constrained, though with some violation when the sample count increased. The improvements in overall constraint loss outweighed the degradation some of the temperature spikes saw when the sample count increased. It did however show that the performance was subpar for the constrained surrogate controller, which shows the importance of the different controller configurations to attain closer control of constraints.

Table 4.7: Loss per controller on the constrained optimization on disturbance set 1. Noise was applied to training and measurements. Two sample sizes were used for training the controllers. Surrogate controller trained on 30% optimal measurements, and was using the t configuration for uc and uc2 controllers.

		500 Samples		2500 Samples	
MS	Controller	Loss	Constraint Loss	Loss	Constraint Loss
-1	uc	21.43	2.51	19.62	1.68
	uc2	24.46	1.10	23.64	1.04
	gc	61.22	0.54	57.83	0.29
-2	uc	18.55	2.53	17.56	0.90
	uc2	21.81	0.97	21.47	0.91
	gc	44.81	0.47	43.89	0.42
-3	uc	23.16	4.40	24.21	4.17
	uc2	31.78	0.80	28.18	0.85
	gc	60.01	0.23	64.04	0.25
-4	uc	18.59	0.85	15.82	0.66
	uc2	22.00	0.68	18.48	0.62
	gc	68.10	0.33	59.15	0.41

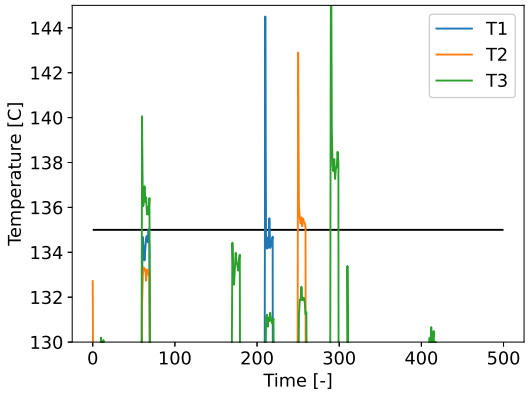


Figure 4.51: MS-1: Stream temperatures and the temperature constraint. Generated using the constrained surrogate controller (uc) with the t configuration. 500 samples used for training.

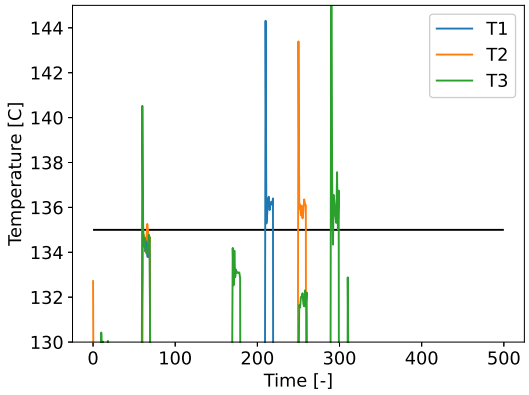


Figure 4.52: MS-1: Stream temperatures and the temperature constraint. Generated using the constrained surrogate controller (uc) with the t configuration. 2500 samples used for training.

4.2.2 Controller configuration

The optimal configuration of the surrogate controller and the mixed controller was investigated. Each run was performed with 2500 samples for training. The performance of the constrained surrogate controller (uc) using different configurations is shown in Table 4.8. The best performing configuration was using MS-4, where the *tv* configuration had the lowest constraint loss of all the modes, but a little larger normal loss than the *t* configuration. MS-2 had normal loss near that of MS-4, but a slightly larger constraint loss. Overall the modes performed similarly, with the *tv* configuration having a slightly lower constraint loss. MS-1 performed best with the $full$ configuration. MS-3 had the largest losses compared to other measurement sets. The *tv* configuration had both the largest normal loss and the lowest constraint loss.

Going forward to the testing of constrained disturbance set 2 for the uc control structure, the $full$ configuration was used for MS-1, all configurations for MS-2, the *tv* configuration for MS-3, and *tv* for MS-4. The reason for testing all configurations of MS-2 is explained in the results for constrained disturbance set 2.

Table 4.8: Loss and constraint loss for the constrained surrogate controller (uc) using disturbance set 1. 2500 samples and noise applied to measurements. Different controller configurations tested.

MS	Configuration	Loss	Constraint Loss
-1	t	19.62	1.68
	tv	20.14	1.30
	full	19.90	0.88
-2	t	17.56	0.90
	tv	17.69	0.78
	full	17.93	0.95
-3	t	24.21	4.17
	tv	26.88	3.12
	full	23.79	3.75
-4	t	15.82	0.66
	tv	17.87	0.31
	full	18.35	0.53

In Table 4.9 the configuration performance is shown for the constrained mixed controller (uc2). Like the constrained surrogate controller (uc), the lowest loss and constraint loss was with MS-4, and the *tv* configuration worked best there. Overall the constraint losses were relatively low for all measurement sets, but MS-4 stood out a little lower. The best performing configuration for MS-2 was the $full$ configuration. MS-3 had relatively high normal losses, but among them the *tv* configuration had the best performance. Additionally,

since the t configuration with MS-4 scored a lower constraint loss, with just a small increase in normal loss, it could also be worth looking into.

Table 4.9: Loss and constraint loss for the constrained mixed controller (uc2) using disturbance set 1. 2500 samples and noise applied to measurements. Different controller configurations tested.

MS	Configuration	Loss	Constraint Loss
-1	t	23.64	1.04
	tv	23.59	1.03
	full	23.30	0.91
-2	t	21.47	0.91
	tv	21.55	0.89
	full	20.48	0.90
-3	t	28.18	0.85
	tv	26.86	0.84
	full	27.91	0.99
-4	t	18.48	0.62
	tv	18.31	0.70
	full	20.05	0.62

Going forward to the testing of constrained disturbance set 2 for the uc2 control structure, the $full$ configuration was used for MS-1, $full$ configurations for MS-2, tv configuration for MS-3, and t and tv for MS-4.

4.2.3 Constrained disturbance set 2

In Table 4.10 we can see that control structure results for disturbance set 2. Starting with MS-2, it was discovered that the constrained surrogate controller (uc) using the *tv* configuration diverged and eventually crashed the simulation. Following up by testing the other configurations, it was observed that the measurement set did not work well with this controller. Without the dynamic back-off scheme (the *t* configuration), the predictions had a relatively good loss with respect to other controllers and measurement sets, but the constrained loss was almost as bad MS-1 using the same controller and configuration. Using the other two configuration modes, where the dynamic back-off scheme activated, a point was reached where the increase in the back-off would cascade and cause further constraint violation. This would continue to snowball until it crashed using the *tv* configuration, while the $full$ configuration did not crash, but did recover very quickly when the disturbances significantly shifted, likely due to the temperature measurement containing reasonable values for the GP to use, unlike the *tv* configuration which only had massively snowballed constraint violation values. The uc controller with the $full$ configuration using MS-2 is shown in Figure 4.60.

Table 4.10: Loss and constraint loss for the different control structures using disturbance set 2. 2500 samples and noise applied to measurements.

MS	Controller (config)	Loss	Constraint Loss
-1	uc (t)	64.48	16.24
	uc2 (full)	75.64	3.63
	gc	79.91	2.57
-2	uc (t)	43.29	13.72
	uc (tv)	-	-
	uc (full)	468.42	13.93
	uc2 (t)	50.67	4.94
	uc2 (full)	55.46	4.98
	gc	89.81	3.11
-3	uc (tv)	33.19	16.75
	uc2 (tv)	33.90	5.39
	gc	242.44	1.44
-4	uc (tv)	25.40	3.96
	uc2 (t)	35.64	3.65
	uc2 (tv)	34.13	3.93
	gc	79.96	3.50

The best performing measurement set was MS-4. The lowest score was achieved with the constrained surrogate controller (uc) using the *tv* configuration. The constrained mixed controller (uc2) showed a smaller constraint loss but significantly larger normal loss than the *tv* configuration. The gradient control structure outperformed the other control structures on constraint loss, but with the worst normal losses compared to the other controllers, aside from the diverging cases of MS-2. MS-3 had relatively good normal loss as well, but the constrained loss was higher than MS-4.

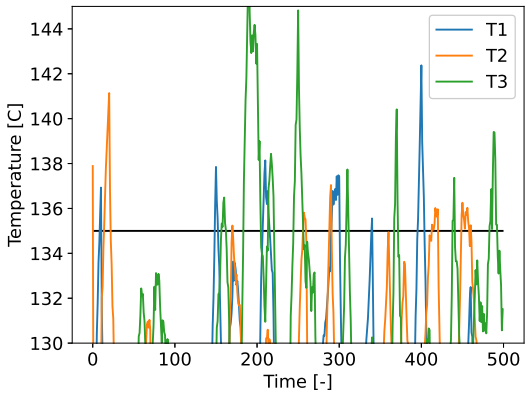


Figure 4.53: MS-1: Temperatures and the temperature constraint. uc controller, *t* configuration.

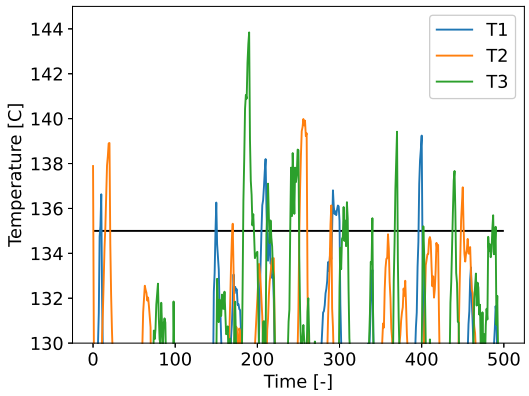


Figure 4.54: MS-1: Temperatures and the temperature constraint. uc2 controller, *both* configuration.

As an example of bad constraint control, the temperatures of the streams with constraints have been plotted, using the uc controller with the *t* configuration and MS-1, and are shown in Figure 4.53. While the temperature constraint was violated, some violations were quickly controlled back down. However, for large and continued disturbance changes into the constrained region, the control structure struggles to counteract that shift. Around the 200 mark, a significant spike above the temperature constraint happened, that was not neutralized. Using the same measurement set, but the uc2 controller with the $full$ configuration, the same peak was adjusted much faster (Figure 4.54). The uc2 controller did not manage to satisfy the constraint every time either with a couple violations over a few time steps around the 250 to 300 mark. The uc2 controller had a larger normal loss than the uc controller, but had a much lower constraint loss.

The constrained gradient control structure (gc) was also under-performing with respect to the normal loss, despite having the best scores for constraint loss. Apart from MS-3 the loss for the gradient control structure was always around 80. The trend is that the constrained gradient control structure managed to control well with respect to the constraint, with most spikes instantly going back down to under the constraint. It is however clear that the control structures struggled with satisfying the constraints when a set of disturbances continued to move into active constraint space, such as around 250 min in Figures 4.55 and 4.56. These plots show how the temperature of the streams changed over time, for MS-1 and MS-2 respectively, using the constrained gradient control structure. As the control structures can not predict the state at the next time step, they are only able to control for the violations at the current time. This led to some unavoidable constraint violation, which turned relatively large even with disturbance set 2, as the change from one testing state to another was relatively short by happening over just 10 time steps. As such, a constraint loss around 3 to 4 can be deemed acceptable.

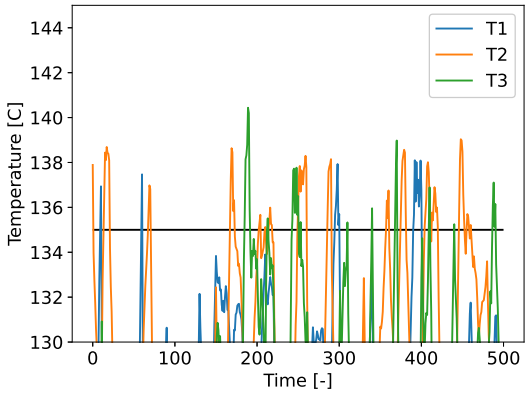


Figure 4.55: MS-1: Temperatures and the temperature constraint, gc controller.

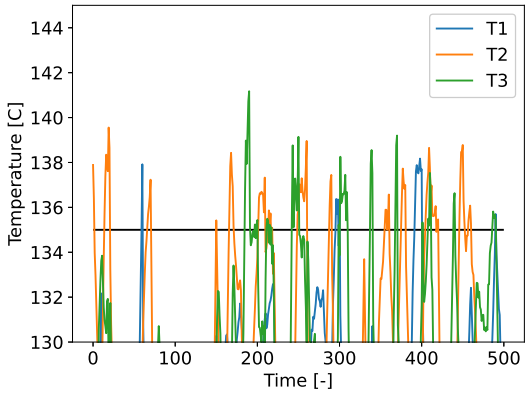


Figure 4.56: MS-2: Temperatures and the temperature constraint, gc controller.

Control using MS-2 performed well in the unconstrained case, but unexpectedly much worse for the constrained case. In Figures 4.57 and 4.59 the valves openings for the t configuration for the surrogate controller (uc) are shown. Likewise, Figures 4.58 and 4.60 show the valves for the $full$ configuration for the same controller. From the figures, especially 4.60, we can see how the valve openings snowball upwards before going back down as the system left the constrained region. While the t configuration did not perform greatly, it did not have the same issue.

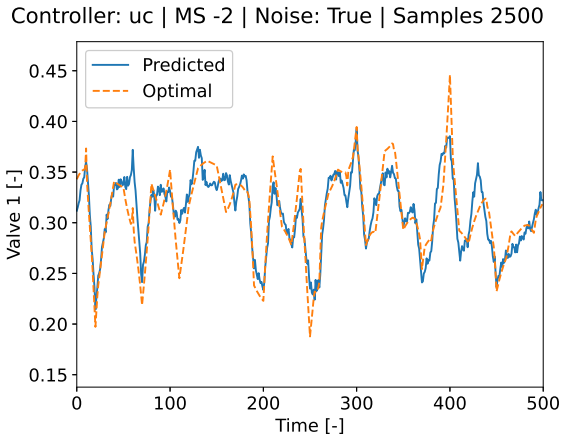


Figure 4.57: MS-2: (Valve 1) uc controller, t configuration.

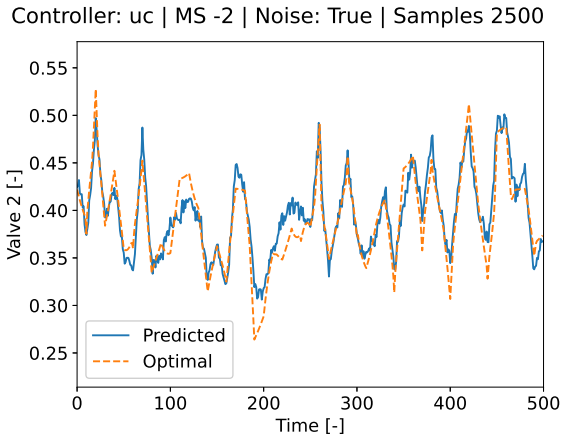


Figure 4.58: MS-2: (Valve 2) uc controller, t configuration.

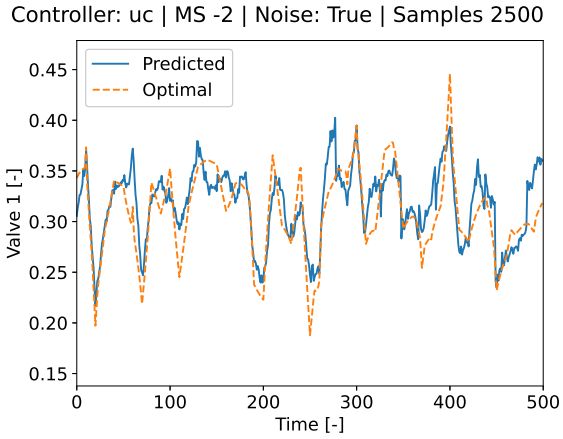


Figure 4.59: MS-2: (Valve 1) uc controller, $full$ configuration.

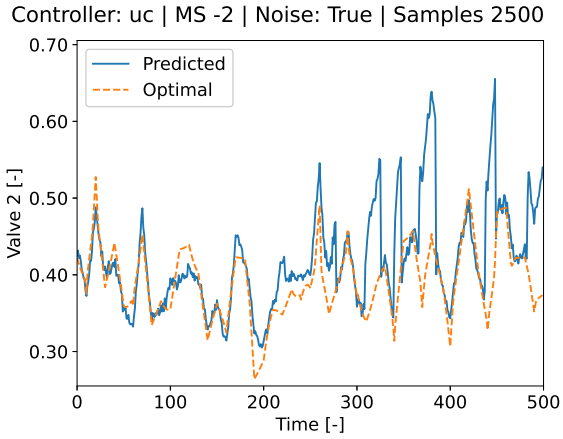


Figure 4.60: MS-2: (Valve 2) uc controller, $full$ configuration.

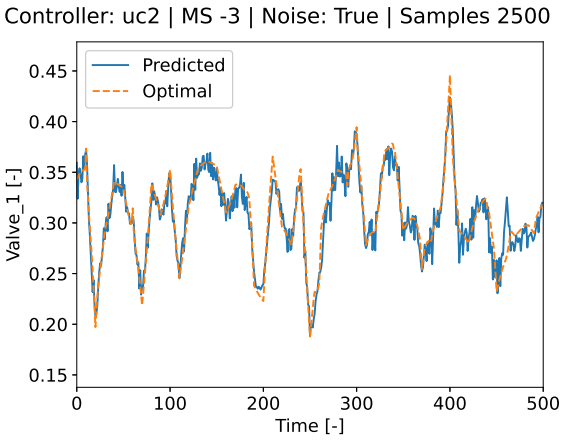


Figure 4.61: MS-3: (Valve 1) uc2 controller, *tv* configuration.

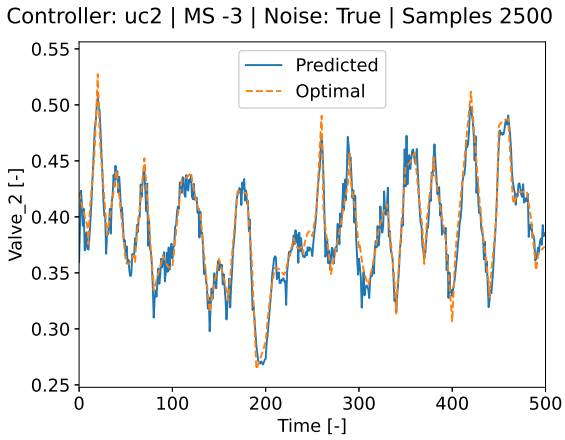


Figure 4.62: MS-3: (Valve 2) uc2 controller, *tv* configuration.

The only controller that achieved good results with MS-3 were the mixed controller using the *tv* configuration, on average just behind losses with MS-4, while having a higher constraint loss. In Figures 4.61 and 4.62, the accurate control of the valves is shown. The cost plot is shown in Figure 4.63, and the temperature and the temperate constraint is shown in Figure 4.64. While the temperature constraint was violated, it was controlled very quickly.

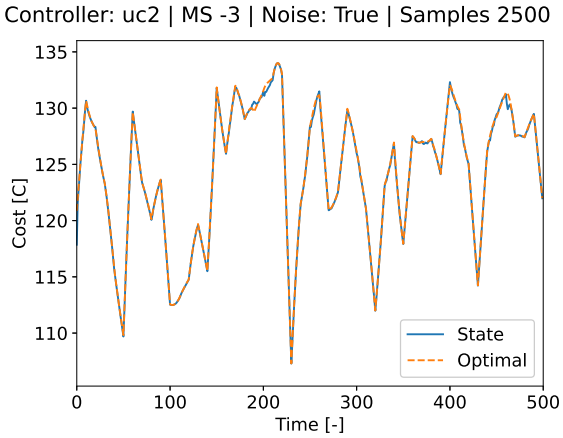


Figure 4.63: MS-3: Temperatures and the temperature constraint, uc2 controller, *tv* configuration.

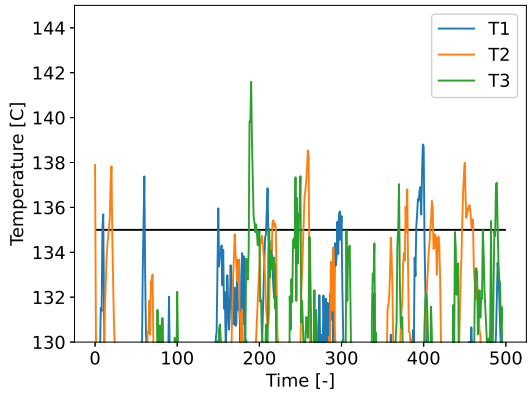


Figure 4.64: MS-2: Temperatures and the temperature constraint, uc2 controller, *tv* configuration.

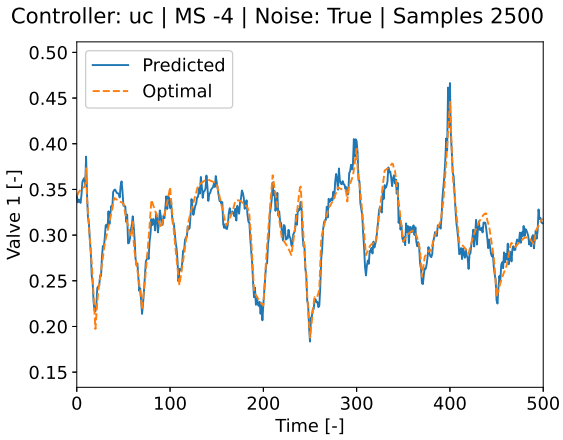


Figure 4.65: MS-4: (Valve 1) uc controller, *tv* configuration.

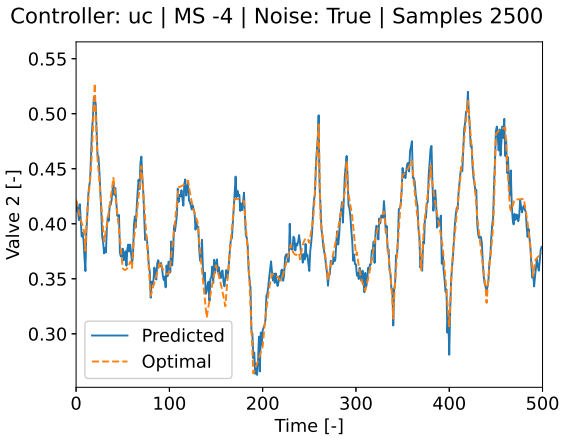


Figure 4.66: MS-4: (Valve 2) uc controller, *tv* configuration.

While several of the controllers performed well on MS-4, the one with the lowest normal loss and low constraint loss was the constrained surrogate controller (uc) using the *tv* configuration. The valve control is shown in Figures 4.65 and 4.66. Like with MS-3, very accurate predictions for the controller was observed, and the constrained loss was very low, with good temperature control as seen in 4.68. From Figure 4.67 it can be seen that the predictions lead to very accurate temperatures throughout the simulation.

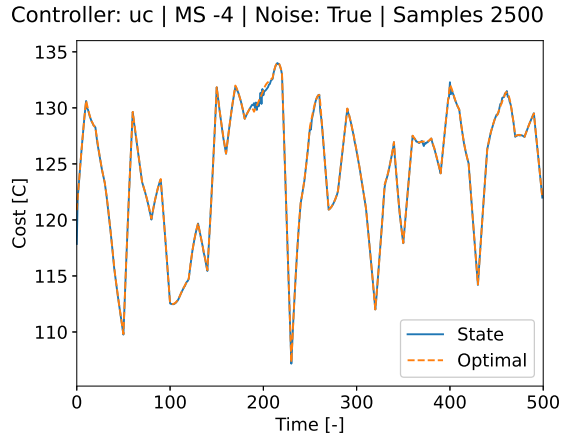


Figure 4.67: MS-3: Temperatures and the temperature constraint, uc2 controller, *tv* configuration.

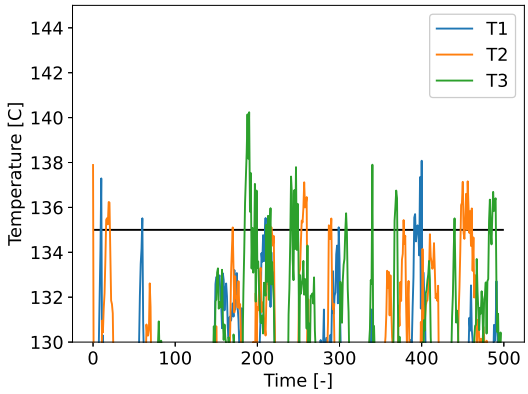


Figure 4.68: MS-2: Temperatures and the temperature constraint, uc2 controller, *tv* configuration.

4.2.4 Noise investigation

Lastly, to investigate the effect of noise on the best configurations for each measurement set. Table 4.11 shows the noise and noise free losses for those configurations. Using MS-1, the constrained gradient control structure (gc) saw some decrease in normal loss when introducing noise, but the constraint loss improved slightly. Both losses increased a little for uc2 controller with the t configuration using MS-2, and the same controller with the tv configuration using MS-3. The configuration that was hurt the most from noise was the constrained surrogate controller (uc) with the tv configuration using MS-4, without noise its performance was significantly better, where the normal loss was much lower than the noisy case. The constraint loss also got reduced when noise was not included. In a noise free case, the uc controller with the tv configuration would be the best choice by far.

Table 4.11: Loss per controller on the constrained optimization on disturbance set 2. Comparing noise vs noise free case. 2500 training samples.

		No noise		Noise	
MS	Controller	Loss	Constraint Loss	Loss	Constraint Loss
-1	gc	74.23	2.71	79.91	2.57
-2	uc2 (t)	48.88	4.86	50.67	4.94
-3	uc2 (tv)	30.99	5.17	33.90	5.39
-4	uc (tv)	17.43	3.29	25.40	3.96

5 Discussion

5.1 Unconstrained case

In terms of results, the best performing controller was the gradient controller on MS1, and then the surrogate controller using MS2 and MS3. MS2 and MS3 both improved when trained with a large percentage of optimal data to train the surrogate controller. With the standard 30% optimal data used, MS3 was the best performing one of the two. MS2 reached the same loss as MS3 when using 70% optimal data, and both performed best with this data split. MS1 and MS2 were more preferable as they only relied upon temperature measurements, unlike MS3 which included the heat capacities of the streams which may be much harder to accurately measure, making it subject to uneven flow rates, measurement noise and inaccuracies.

The gradient control structure performed most favorably with MS1, and seemed to respond better to the different disturbances overall. As a downside, its nature for convergence may make it slower than the surrogate controller, but that was not a problem in this study. The surrogate controller makes one prediction and steps directly to the predicted optimal, with exception for the smoothing factor aimed to reduce large sensitivity to noise. The gradient control structure, on the other hand, keeps updating the valve splits and waiting for new predictions until the gradient is near or at zero. This is particularly detrimental when there are large step changes in the disturbances, such as presented in disturbance set 1 using MS1. This is not expected to be a large problem in practice where disturbances do not change as much over a single time step, such as with disturbance set 2. The gradient control structure outperformed the surrogate controller on disturbance set 2, which supports this claim.

In terms of disturbances, MS1, MS2 and MS4 in general responded well to changes in temperature disturbances, but did less favorably with changes in the heat capacities and overall heat transfer coefficients. MS3 on the other hand dealt better heat capacities but worse with the other disturbances. The exception was the gradient control structure using MS1, which responded well to most disturbances, and had the best performance. This may be related to the Jäschke temperature [10], which is used for near-optimal operation of heat exchanger networks such as in this case study. The Jäschke temperature would in this case depend on the input and output of the cold stream, as well as the input hot stream. When the Jäschke temperature is kept equal for all split streams, the largest heat transfer can be achieved. The difference in Jäschke temperatures for each stream can then be controlled to zero for optimal operation, similarly to the role of the gradients in this control structure. Since the same information is available in MS1, then it is possible that the Gaussian process is able to use that for predictions.

Overall, the gradient control structure (g) responded to more disturbances, which was clearly seen with MS1, but the same actions was also seen with MS2 and MS3, except that the predictions were not good. This stronger response makes it more fit to use in system where other disturbances than the temperatures change at a relatively high rate. The surrogate controller on the other hand dealt well mostly with temperatures, and

heat capacities if provided to the controller. The surrogate controller was more sensitive to the distributions of the measurement with respect to the optimal valve openings. With some measurements set it performed much better having more training samples around the optimum, while for other it would show degraded performance when too much optimal data was used.

5.2 Constrained case

Unlike the unconstrained case, MS-4 was the best performing controller of them all, with MS-3 getting close to that as well. However as already mentioned, its reliance on heat capacities makes it less ideal for real world use. As such, the better performance from MS-4 was ideal.

For the constrained case, the surrogate controller (uc) and mixed controller (uc2) had good performance for MS-3 and MS-4, and greatly outperformed the other measurement sets in terms of normal loss. The constrained gradient control structure (gc) suffered relatively high normal losses, but had the lowest constrained loss. Given the good performance of the constrained gradient control structure with MS1 in the unconstrained case, it is surprising that it did not stand out in the constrained case. It may be that the selector scheme could have its weights tuned more extensively to increase the performance to be on a competitive level with the other controllers.

It is interesting to note that with the introduction of stream temperatures in the negatively marked measurement sets, where MS-1 is the same as MS1, that the performance is so different from MS-3 and MS-4 compared to their previous counterparts. MS-4 is effectively MS-1 but with the valve splits included, which suggest that they have a large effect on the predictions when it comes the constrained case. Similarly to the unconstrained case, the Jäschke temperature may be at effect here. It is however interesting how the performance of MS-1 and the constrained gradient control structure (gc) was worse than the other controllers, based on the performance in the unconstrained case. Both MS-1 and MS-4 contained the measurements that the Jäschke temperature depends on, due to the cold output stream temperatures being introduced in all the constrained measurement sets. With MS-4 however, the surrogate controller (uc) was the controller that had the best performance, while the constrained gradient control structure (gc) was almost unchanged with relation to MS-1.

The presence of noise also worked out well, while generally some gains were to be made without noise, using temperature measurements did not cause larger issues with the predictions, to the point that the predictions looked overall better for the noisy constrained plots compared to the unconstrained plots without noise. The constrained surrogate controller with *tv* configuration, using MS-4, was however greatly affected by noise. It was still the best performing configuration overall, but it appeared that it could perform even better without the noise.

In general, the constrained gradient control structure (gc) outperformed the other controllers with constraint loss. This was likely because the controller selection structure worked well when entering the constrained

space. The constrained gradient control structure did have the higher normal loss in general, which suggests the predictions were not as good compared to the other controllers when not working in the constrained region. The constrained surrogate controller (uc) had both the best and worst performance, using the tv configuration. With MS-4, it worked without much issue, and the dynamic back-off scheme helped it out compared to without it. With MS-2, the dynamic back-off seemed to cause the opposite reaction, once the controller violated the constraint the increase in back-off could cause valve openings to react in the opposite direction, which then compounded with more back-off which quickly led the temperature violation variables (such as T3v) away from realistic values. This was particularly dangerous since these variables reached values far from anything the Gaussian predictor was trained on, which can produce irregular results. This risk should be considered when implanting such a basic dynamic back-off, and possibly add safeguards to prevent the temperature violations from reaching too large values.

The possible reason that the dynamic back-off failed could be that this measurement set mainly consisted of temperatures from the hot streams along with the temperature constraint variables (for the tv configuration). Most of the controller action was with respect to the hot streams measurement, and when the back-off changed to something large, the system could not deal with it. With the $full$ configuration, the actual temperatures were present so when the disturbances changed enough, to the point where the system was no longer in the constrained space, it had some normal values to use to control itself back to normal operation. When removing the dynamic back-off (the t configuration) it was much more well behaved, but did not manage to control for the temperature constraint.

The constrained mixed control (uc2) was the most consistent controller, all uses had constraint loss in the range from 3 to 5. The normal loss varied between the measurement sets, but did not change much between the controller configurations for a measurement set. It did not significantly change performance from the introduction of the temperature violation variables either.

6 Conclusions

The use of GP to control networks of heat exchangers is possible, including when in the presence of active constraints on the temperature of the streams. For an unconstrained case, predicting gradients with GP with MS1, and then using a standard gradient (setpoint) controller gives the best performance. MS1 measured the input and output temperatures of the cold stream, and the input temperature of the hot stream. However, it is possible to train the surrogate controller to directly predict optimal valve openings with performance close to that, using MS2. MS2, like MS1, only contained temperatures, making it a suitable candidate for predictions. A surrogate controller using MS3 had similar performance to the same controller using MS2, but MS3 relied on heat capacities, which in the common case makes it less ideal to measure, and is thus not suggested when the other measurement sets outperform or match its prediction performance.

When adding a constraint, the problem became slightly harder, and the sample size was increased to give the GP module more info to work from. The best performance was obtained with the surrogate controller (uc) with the *tv* configuration, and using MS-4. MS-4 contained the same temperature measurements as MS-1/MS1, but also the current valve openings. In the constrained case it outperformed MS-1, and scored a low constraint loss and the lowest normal loss for disturbance set 2. The mixed controller (uc2) with the t configuration had a higher normal loss but slightly lower constraint loss using MS-4. MS-3 was second best overall, but as mentioned, relied on heat capacities. It also had a higher constraint loss compared to MS-4.

In the unconstrained case, the gradient controller responded to more disturbances than the surrogate controller which mainly reacted to changes in temperatures. The surrogate controller relied more strongly on the training data distribution, where some measurement sets showed better performance with more measurements near optimal operation, while others favored having more randomly distributed measurements. In the constrained case, the constrained gradient control structure had best control at the constraint, attaining the lowest constraint loss. The constrained surrogate controller was strongly dependent on the measurement set, where one greatly improved its performance where another caused it to diverge due to reacting in the opposite direction of the optimum. The constrained mixed controller had the most stable performance, but did not outperform the other control structures.

6.1 Future work

A better analysis could have been attained if disturbance set 2 for the two cases were just one and the same. Comparing the performance of the the unconstrained and constrained could have provided more interesting observations of performance changes when introducing the temperature constraint and the new control structures. A more realistic disturbances set 2 could also have been made, with less frequent and large changes in disturbances, along with regions of stable operation with slow and gradual changes. Interaction with those changes while at the temperature constraint for one or two streams, could give more insight on how the controllers act around the constraint, and how well multiple active constraints are handled.

The good performance of MS-4 is something that could be investigated further, as it scored much lower loss than the other measurement sets when using the constrained surrogate controller (uc). That and the performance of MS1 with the gradient controller (g), may indicate useful measurement sets, which is justified given that they included the variables used for calculating the Jäschke temperatures.

References

[1] Carl Edward Rasmussen and Williams Christopher K I. *Gaussian processes for machine learning*. MIT Press, 2008.

[2] J.J.J. Chen. Logarithmic mean: Chen's approximation or explicit solution? *Computers & Chemical Engineering*, 120:1–3, 2019. ISSN 0098-1354. doi: https://doi.org/10.1016/j.compchemeng.2018.10.002. URL `https://www.sciencedirect.com/science/article/pii/S0098135418309797`.

[3] Alexander Forrester, Andras Sobester, and Andy Keane. *Engineering Design Via Surrogate Modelling: A Practical Guide*. John Wiley & Sons, Ltd, 07 2008. ISBN 978-0-470-06068-1. doi: 10.1002/9780470770801.

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

[5] Sigurd Skogestad. Self-optimizing control: the missing link between steady-state optimization and control. *Computers & Chemical Engineering*, 24(2):569–575, 2000. ISSN 0098-1354. doi: https://doi.org/10.1016/S0098-1354(00)00405-1. URL `https://www.sciencedirect.com/science/article/pii/S0098135400004051`.

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

[7] Dinesh Krishnamoorthy. *Novel Approaches to Online Process Optimization under Uncertainty*. PhD thesis, NTNU, November 2019.

[8] Dinesh Krishnamoorthy and Sigurd Skogestad. Online process optimization with active constraint set changes using simple control structures. *Industrial & Engineering Chemistry Research*, 58(30):13555–13567, 2019. doi: 10.1021/acs.iecr.9b00308. URL `https://doi.org/10.1021/acs.iecr.9b00308`.

[9] Dinesh Krishnamoorthy and Sigurd Skogestad. Systematic design of active constraint switching using selectors. *Computers & Chemical Engineering*, 143:107106, 2020. ISSN 0098-1354. doi: https://doi.org/10.1016/j.compchemeng.2020.107106. URL `https://www.sciencedirect.com/science/article/pii/S0098135420307274`.

[10] 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 Data tables

A.1.1 Unconstrained data

Table A.1: Performance of controllers on the unconstrained optimization on disturbance set 1. No noise in the system. Two sample sizes were used for training the controllers. Surrogate controller trained on 70% optimal measurements.

MS	Controller	500 Samples	2500 Samples
1	u	18.27	19.08
	g	23.84	17.15
2	u	13.10	11.85
	g	55.80	40.42
3	u	11.85	11.83
	g	61.79	48.06
4	u	31.53	132.26
	g	46.04	32.00

Table A.2: Loss per controller on the unconstrained optimization on disturbance set 2. No noise. Two sample sizes were used for training the controllers. Surrogate controller trained on 30% optimal measurements.

MS	Controller	500 Samples	2500 Samples
1	u	118.76	116.88
	g	54.94	50.94
2	u	80.76	68.41
	g	298.29	294.84
3	u	67.93	59.68
	g	305.23	294.84
4	u	122.07	119.40
	g	227.11	178.50

Table A.3: Loss using gradient controller for the unconstrained optimization on disturbance set 2. No noise. Two sample sizes were used for training the controllers. Only with MS1 did the loss decrease with higher integral gain.

MS	Controller	500 Samples	2500 Samples
1	g	54.94	50.94
2	g	352.36	314.25
3	g	344.38	334.95
4	g	209.72	156.99

A.2 Unconstrained gradients

Here gradient control for the gradient controller (g) using disturbance set 1, 500 samples.

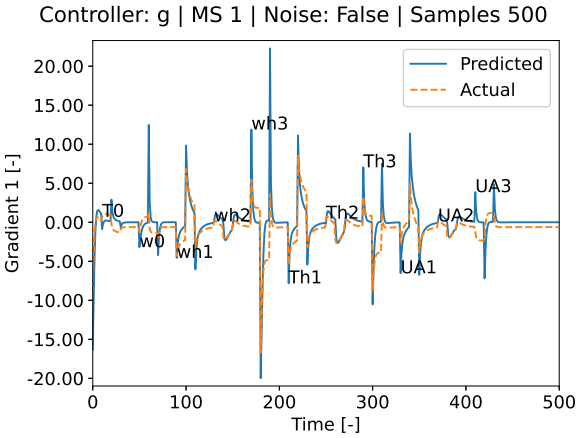


Figure A.1: MS1: (Gradient 1) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

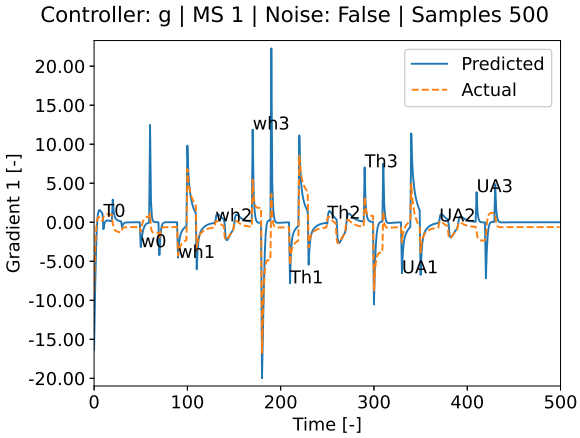


Figure A.2: MS1: (Gradient 2) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

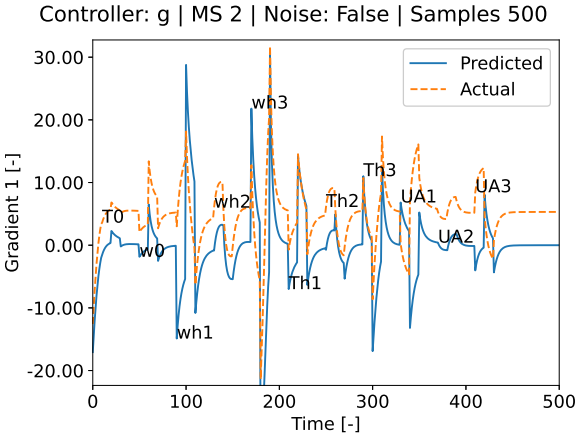


Figure A.3: MS2: (Gradient 1) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

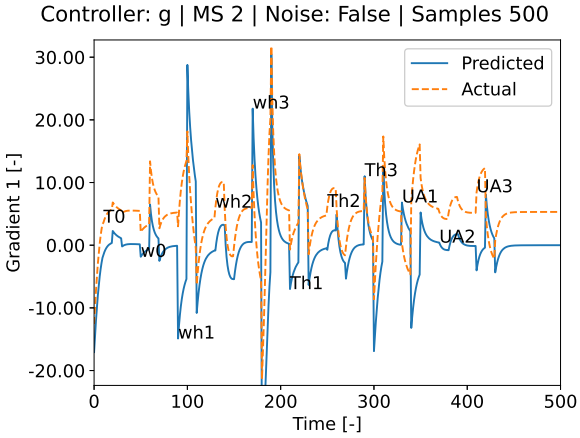


Figure A.4: MS2: (Gradient 2) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

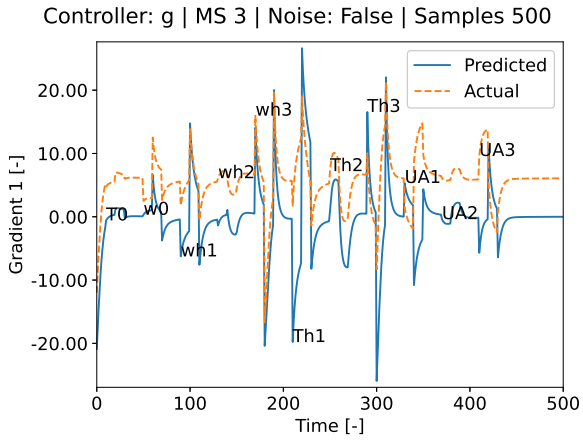


Figure A.5: MS3: (Gradient 1) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

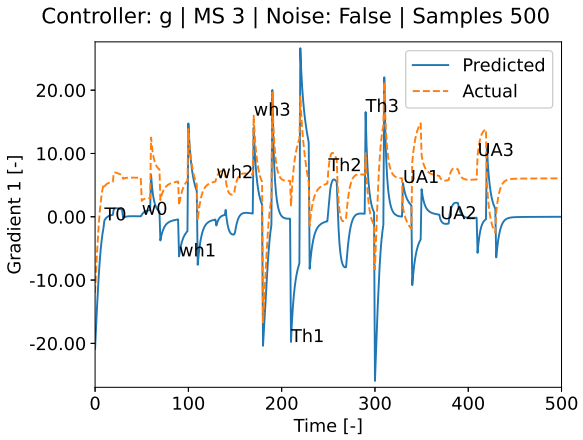


Figure A.6: MS3: (Gradient 2) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

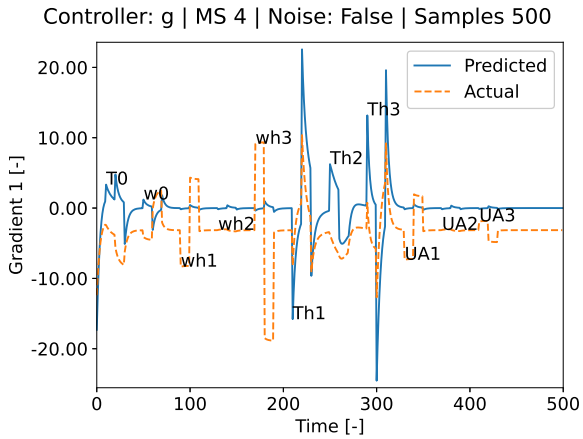


Figure A.7: MS4: (Gradient 1) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

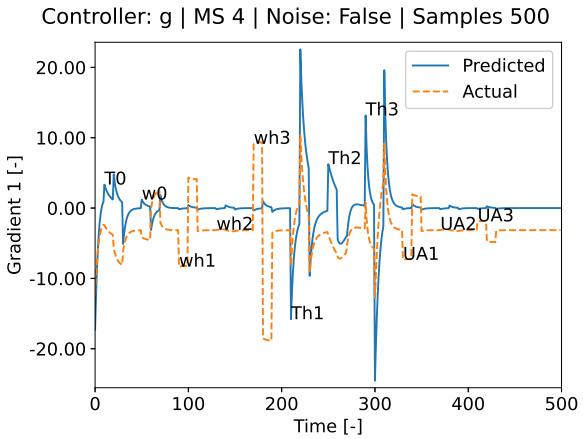


Figure A.8: MS4: (Gradient 2) Gradient predictions using disturbance set 1. Changed disturbances are marked, where a disturbance first has a positive change, and then a negative change afterwards.

B Code

B.1 The plant

The following code contains the model used for make the training data and generating the optimal information during the simulation.

```
import numpy as np
from casadi import *

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 = {
    -4: ['T0', 'T', 'T1', 'T2', 'T3', 'Th1', 'Th2', 'Th3', 'alpha1', 'alpha2'],
    -3: ['T0', 'T', 'T1', 'T2', 'T3', 'The1', 'The2', 'The3', 'w0', 'wh1', 'wh2', 'wh3'],
    -2: ['T0', 'Th1', 'Th2', 'Th3', 'The1', 'The2', 'The3', 'T1', 'T2', 'T3'],
    -1: ['T0', 'T1', 'T2', 'T3', 'Th1', 'Th2', 'Th3'],
    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 {'x': x, 'par': par, 'y': y, 'T': x[x_vars.index('T')], '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']}
```

B.2 Data generation

For the unconstrained case, two data generation scripts were made, one for each controller.

Gradient data:

```
import sys

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 ∗ 20), axis=0)
    x = np.concatenate([np.zeros((1, N ∗ 20)), x, np.ones((1, N ∗ 20))], 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), 10 * N)
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(10 * N)]

# Generating measurements, priors and targets
u_span = []
d_span = []
J_span = []
J_span_chen = []
grad_span_chen = []
grad_span = []

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

    u = alpha[:, i]
    # print(u)
    if u[0] > 0.65 or u[0] < 0.1:
        print('u0 too big/small')
        errors += 1
        i += 1
        continue
    if u[1] > 0.65 or u[1] < 0.1:
        print('u2 too big/small')
        errors += 1
        i += 1
        continue
    if np.sum(u) > 0.95:
        print(f'sum u too large')
        errors += 1
        i += 1
        continue

    # Calculate optimal output temp from optimal u
    try:
        cost = hex3.cost(u, copy.deepcopy(params))
        cost_chen = hex3_chen.cost(u, copy.deepcopy(params))

        if not cost['success'] or not cost_chen['success']:
            errors += 1
            i += 1
            print('Bad hex3 cost, errors: ', errors)
            continue
        gradient_chen = hex3_chen.grad(u, copy.deepcopy(params))
        grad = hex3.grad(u, copy.deepcopy(params))

        if not grad['success']:
            errors += 1
            i += 1
            print('Bad hex3 grad, errors: ', errors)
            continue

    except Exception as e:
        errors += 1
        import traceback
        print(f'Error "{e}"', file=sys.stderr)
        i += 1
        print('Skipping idx', i)
        continue
    else:
```

```
            print('Success solutions: ', finished + 1)
            i += 1
            finished += 1

        # Save values
        u_span.append(np.array(u))

        d_span.append(params)
        J_span.append(-cost['J'][0])
        J_span_chen.append(-cost_chen['J'][0])
        grad_span.append(grad['grad'])

        grad_span_chen.append(gradient_chen['grad'])

    u_span = np.array(u_span)
    d_span = np.array(d_span, dtype=dict)
    J_span = np.array(J_span)
    J_span_chen = np.array(J_span_chen)
    grad_span_chen = np.array(grad_span_chen)
    grad_span = np.array(grad_span)

    return u_span, d_span, J_span, J_span_chen, grad_span_chen, grad_span


def save_data(name, u_span, d_span, J_span, J_span_chen, g_span_chen, g_span):
    u_headers = [f'u{i}' for i in range(u_span.shape[1])]
    g_headers_chen = [f'gc{i}' for i in range(g_span_chen.shape[1])]
    g_headers = [f'g{i}' for i in range(g_span_chen.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)})

    J_span_pd = pd.DataFrame.from_dict({J_header: J_span})
    J_span_chen_pd = pd.DataFrame.from_dict({J_chen_header: J_span_chen})
    d_span_pd = pd.DataFrame.from_records(d_span)

    g_span_chen_pd = pd.DataFrame.from_dict({key: val for key, val in zip(g_headers_chen, g_span_chen.T)})

    g_span_pd = pd.DataFrame.from_dict({key: val for key, val in zip(g_headers, g_span.T)})

    frames = pd.concat([u_span_pd, d_span_pd, J_span_pd, J_span_chen_pd, g_span_pd, g_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],
                d=frames.iloc[:, 2:-6],
                J=frames.iloc[:, -6:-5],
                J_chen=frames.iloc[:, -5:-4],
                g=frames.iloc[:, -4:-2],
                gc=frames.iloc[:, -2:])

    return data


if __name__ == '__main__':

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

        np.random.seed(2025)
        u_span, d_span, J_span, J_span_chen, gradient_span_chen, gradient_span = gen_dataset(samples, 1)
        save_data(f'.\\datasets\\new_train_gradient{samples}.csv', u_span, d_span, J_span, J_span_chen, gradient_span_chen,
                  gradient_span)
```

```
        print('Done')

    # Test set
    print('Generating test data....')
    np.random.seed(2028)
    u_span, d_span, J_span, J_span_chen, gradient_span_chen, gradient_span = gen_dataset(samples, 1.35)
    save_data(f'.\\datasets\\new_test_gradient{samples}.csv', u_span, d_span, J_span, J_span_chen, gradient_span_chen,
              gradient_span)
    print('Done')
```

Optimal valve data:

```
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))
    # Note: hex3 model data not used for predictions
    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')
```

In the constrained case, data generation scripts were made for the surrogate controller (uc) and the mixed controller (uc2), since they relied upon the predictions for valve splits that take took the temperature into consideration. The constrained gradient controller (gc) however, used the same data as the unconstrained gradient controller (g) since the gradients do not respect the constraint.

Constrained optimal valve data:

```
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 = []

hex3_chen.Ti_max = 135
hex3.Ti_max = 135

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

    if any(alpha[:, i] < 0.05) or np.sum(alpha[:, i]) > 0.95:
        errors += 1
        i += 1
        print('Too low or high alphas: ', errors)
        continue

    u_chen = hex3_chen.optim(copy.deepcopy(params))
    # Note: hex3 model data not used for predictions
    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]:
        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_constrained_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_constrained_prediction{samples}.csv', u_span, u_span_chen, u_rand_span, d_span, J_span,
            J_span_chen)
    print('Done')
```

B.3 Gaussian implementation

```
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()
        np.random.seed(2311)

        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((-1, 1))
            Y_train = self.norm_y.fit_transform(Y_train)
            # print(norm_y.min_)
```

```
            # print(norm_y.scale_)
            # print(norm_y.feature_range)
        # print(X_train.shape)
        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])
        # print(Y_train.shape)
        icm = GPy.util.multioutput.ICM(input_dim=X_train.shape[1], num_outputs=Y_train.shape[1], kernel=K)
        # print(np.array)
        self.m = GPy.models.GPCoregionalizedRegression([X_train, X_train], Y_train, kernel=icm)

        if num_restarts:
            # self.m.optimize(messages=True)
            self.m.optimize_restarts(messages=True, num_restarts=num_restarts)
        print(self.m)
        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.4 Control structure implementations

Surrogate control structure:

```
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel
from optimal_u.hex3_gen_u_optim import load_data


def measurements_to_array(u, meas, d_order):
    # Distubances
    if isinstance(meas, pd.DataFrame):
        meas = meas.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u_u0 = u['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]
```

```
    # 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')

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

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

    # Inputs
    if isinstance(meas, np.ndarray):
        if toggle:
            # Move
            X = np.array([np.append(np.array([u_u0, u_u1]), meas[:-2])])
        else:
            X = np.array([meas])
    else:
        X = [*([u_u0, u_u1] * toggle), *[meas[k] for k in d_keys]]
        X = np.array([X], dtype=np.float)
    return X


def disturbance_to_array(u, d, d_order):
    # 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['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    X = [*([u_u0, u_u1] * toggle), *[d[k] for k in d_keys]]
    X = np.array([X], dtype=np.float)
    return X


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')

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

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

    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 generate_meas_set_data(mset, u, d):
    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(hex3_chen.meas_sets[mset]):
            res_dict[key] = result['y'][idx]
        results.append(res_dict)
    return np.array(results, dtype=dict)


class InputController:
    def __init__(self, training_database, meas_set=None, noise=False):
        train_data = load_data(training_database)

        if meas_set is not None:
            d_keys = hex3_chen.meas_sets[meas_set]
            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(0.30 * len(u1)) # 0.7 means 30% non-optimal
            u3 = np.concatenate((u1[:split] + diff[:split], u1[split:] + diff[split:] * 0.5))
            # u3 = u1 # only random
            # u3 = u2 # only optimal
            disturbances_train = generate_meas_set_data(meas_set, u3, train_data['d'])

        else:
            raise Exception('Not supported for this controller')
            # d_keys = None
            # disturbances_train = train_data['d']
            # u3 = train_data['u_rand']

        X_train, Y_train, key_order = preprocess(train_data['u_chen'], disturbances_train, d_order=d_keys,
                                                 u_input=u3, add_noise=noise)
        # print(X_train[0])
        # print(Y_train[0])
        self.outputs = len(Y_train[0])
        self.real_cv = np.empty((0, 2), int)
        self.t_samples = len(Y_train)
        self.meas_set = meas_set
        self.d_keys = key_order
        self.gp = GPModel()
        self.gp.start()
        self.gp.train(X_train, Y_train, num_restarts=1, normalize_y=True)

    def __str__(self):
        return 'u'

    def plot_init(self, axes):
        lines = []
        rlines = []
        for idx, ax in enumerate(axes):
            ax.set_ylabel(f'Valve {idx + 1} [-]')
            ax.set_xlabel('Time [-]')
            ax.set_ylim([0.25, 0.35])
            ax.set_xlim([0, 1])
            line, = ax.plot([], [], label='Predicted')
            lines.append(line)
            rline, = ax.plot([], [], '--', label='Optimal')
            rlines.append(rline)
            ax.legend()
        return lines, rlines

    def gen_real_cv(self, u, u_opt, d):
        self.real_cv = np.append(self.real_cv, np.array([u_opt]), axis=0)

    def plot_prediction(self, axes, lines, time, variables, real_lines):
        for idx, (ax, line) in enumerate(zip(axes, lines)):
```

```
            ylim = ax.get_ylim()
            # print(ylim)
            if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(variables[:, idx])-0.05, np.max(variables[:, idx])+0.05])
            ylim = ax.get_ylim()
            if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(self.real_cv[:, idx])-0.05, np.max(self.real_cv[:, idx])+0.05])

            # Fix for auto-plotting
            if self.meas_set == 3 and idx == 0:
                ax.set_ylim([ylim[0], 0.38])
            ax.set_xlim([0, len(time)])
            line.set_xdata(time)
            line.set_ydata(variables[:, idx])
            real_lines[idx].set_xdata(time)
            real_lines[idx].set_ydata(self.real_cv[:, idx])

    def predict(self, u, measurement):
        # print('.-'*30)
        # print(u, measurement)
        try:
            if self.meas_set is not None:
                X = measurements_to_array(u, measurement, self.d_keys)
            else:
                X = disturbance_to_array(u, measurement, self.d_keys)
                raise Exception('Unhandled')
                # print(X)
        except:
            self.close()
            raise
        u_old = u
        alpha = 0.95
        u, u_var, _ = self.gp.test(X) # print(f'Before {u} | grad {gradients}')
        u = u[0, :]
        # print(u)
        return alpha*u + (1. - alpha)*np.array(u_old), alpha*u + (1. - alpha)*np.array(u_old)

    def __del__(self):
        self.close()

    def close(self):
        self.gp.exit()


if __name__ == '__main__':
    # train = f'..\\gradient\\datasets\\train_gradient{500}.csv'
    # controller = GradientController(train)
    # controller.close()
    # print('Controller closed')
    pass
```

Gradient control structure:

```
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel
from gradient.gradient_gen_data import load_data


def measurements_to_array(u, meas, d_order):
    # Distubances
    if isinstance(meas, pd.DataFrame):
        meas = meas.to_dict('records')

    if isinstance(u, pd.DataFrame):
```

```
        u_u0 = u['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    if isinstance(meas, np.ndarray):
        if toggle:
            # Move valve openings to start
            X = np.array([np.append(np.array([u_u0, u_u1]), meas[:-2])])
        else:
            X = np.array([meas])
    else:
        X = [*([u_u0, u_u1] * toggle), *[meas[k] for k in d_keys]]
        X = np.array([X], dtype=np.float)
    return X


def disturbance_to_array(u, d, d_order):
    # 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['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    X = [*([u_u0, u_u1] * toggle), *[d[k] for k in d_keys]]
```

```
    X = np.array([X], dtype=np.float)
    return X


def preprocess(u: pd.DataFrame, d: [pd.DataFrame, np.ndarray], g: [pd.DataFrame, np.ndarray], d_order=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['u0']
        u_u1 = u['u1']
    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')

    # TODO: Just use original variables
    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
    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 gradients values
    if isinstance(g, pd.DataFrame):
        g0 = g['gc0']
        g1 = g['gc1']
    else:
        raise Exception('Unhandled datatype for gradients')

    Y = np.array(list(zip(g0, g1)))

    return X, Y, d_order


def generate_meas_set_data(mset, u, d):
    d = d.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u = np.array(list(zip(u['u0'], u['u1'])))

    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(hex3_chen.meas_sets[mset]):
            res_dict[key] = result['y'][idx]
        results.append(res_dict)
    return np.array(results, dtype=dict)


class GradientController:
    def __init__(self, training_database, meas_set=None, noise=False):
        train_data = load_data(training_database)
        u = train_data['u']
        d = train_data['d']
        g = train_data['gc']

        if meas_set is not None:

            d_keys = hex3_chen.meas_sets[meas_set]
            # print('Solving measurment set data...')
            disturbances_train = generate_meas_set_data(meas_set, u, d)

        else:
            d_keys = None
            disturbances_train = train_data['d']

        X_train, Y_train, key_order = preprocess(u, disturbances_train, g, d_order=d_keys,
                                                 add_noise=noise)
        # print(X_train[0])
        # print(Y_train[0])
        self.outputs = len(Y_train[0])
        self.t_samples = len(Y_train)
        self.real_cv = np.empty((0, 2), float)

        self.meas_set = meas_set
        self.d_keys = key_order
        self.gp = GPModel()
        self.gp.start()
        self.gp.train(X_train, Y_train, num_restarts=1, normalize_y=True)

    def __str__(self):
        return 'g'
    def plot_init(self, axes):
        lines = []
        rlines = []
        for idx, ax in enumerate(axes):
            ax.set_ylabel(f'Valve {idx + 1} [-]')
            ax.set_xlabel('Time [-]')
            ax.set_ylim([0.25, 0.35])
            ax.set_xlim([0, 1])
            line, = ax.plot([], [], label='Predicted')
            lines.append(line)
            rline, = ax.plot([], [], '--', label='Optimal')
            rlines.append(rline)
            ax.legend()
        return lines, rlines



    def gen_real_cv(self, u, u_opt, d):
        self.real_cv = np.append(self.real_cv, np.array([u_opt]), axis=0)
```

```
def plot_prediction(self, axes, lines, time, variables, real_lines):
    for idx, (ax, line) in enumerate(zip(axes, lines)):
        ylim = ax.get_ylim()
        # print(ylim)
        if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
            ax.set_ylim([np.min(variables[:, idx])-0.05, np.max(variables[:, idx])+0.05])
        ylim = ax.get_ylim()
        if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
            ax.set_ylim([np.min(self.real_cv[:, idx])-0.05, np.max(self.real_cv[:, idx])+0.05])

        # Fix for auto-plotting
        if self.meas_set == 1 and idx == 0:
            ax.set_ylim([ylim[0], 0.38])
        if self.meas_set == 3 and idx == 0:
            ax.set_ylim([ylim[0], 0.38])
        if self.meas_set == 4 and idx == 0:
            ax.set_ylim([ylim[0], 0.38])
        ax.set_xlim([0, len(time)])
        line.set_xdata(time)
        line.set_ydata(variables[:, idx])
        real_lines[idx].set_xdata(time)
        real_lines[idx].set_ydata(self.real_cv[:, idx])

# def gen_real_cv(self, u, u_opt, d):
# real_g = hex3_chen.grad(u, d)
# assert real_g['success']
# # print(real_g)
# self.real_cv = np.append(self.real_cv, np.array([real_g['grad']]), axis=0)
# # print(self.real_cv)
#
# def plot_init(self, axes):
# lines = []
# rlines = []
# for idx, ax in enumerate(axes):
# ax.set_ylabel(f'Gradient {idx + 1} [-]')
# ax.set_xlabel('Time [-]')
# ax.set_ylim([-5, 5])
# ax.set_xlim([0, 1])
# line, = ax.plot([], [], label='Predicted')
# lines.append(line)
# rline, = ax.plot([], [], '--', label='Actual')
# rlines.append(rline)
# ax.legend()
# return lines, rlines
#
# def plot_prediction(self, axes, lines, time, variables, real_lines):
# for idx, (ax, line) in enumerate(zip(axes, lines)):
# ylim = ax.get_ylim()
# # print(ylim)
# if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
# ax.set_ylim([np.min(variables[:, idx])-1), np.max(variables[:, idx])+1])
# if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
# ax.set_ylim([np.min(self.real_cv[:,idx])-1, np.max(self.real_cv[:,idx])+1])
# ax.set_xlim([0, len(time)])
# # print(time)
# # print(variables[:, idx])
# line.set_xdata(time)
# line.set_ydata(variables[:, idx])
# real_lines[idx].set_xdata(time)
# real_lines[idx].set_ydata(self.real_cv[:, idx])

def predict(self, u, measurement):
    # print('.-'*30)
    # print(u, measurement)
    try:
        if self.meas_set is not None:
            X = measurements_to_array(u, measurement, self.d_keys)
```

```
            else:
                X = disturbance_to_array(u, measurement, self.d_keys)
                # print(X)
        except:
            self.close()
            raise

        gradients, grad_var, _ = self.gp.test(X) # print(f'Before {u} | grad {gradients}')
        # k = -0.0005
        k = -0.0010 if self.meas_set == 1 else -0.0005
        u = u + np.array([k * gradients[0][0], k * gradients[0][1]])

        return u, u # np.copy(gradients[0, :])

    def __del__(self):
        self.close()

    def close(self):
        self.gp.exit()


if __name__ == '__main__':
    train = f'..\\gradient\\datasets\\train_gradient{500}.csv'
    controller = GradientController(train)
    controller.close()
    print('Controller closed')
```

Constrained surrogate control structure:

```
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel
from optimal_u_constrained.hex3_gen_u_optim import load_data


def measurements_to_array(u, meas, d_order, meas_set, meas_config, last_run=None):
    # Distubances
    if isinstance(meas, pd.DataFrame):
        meas = meas.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u_u0 = u['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

    d_keys = d_order.copy()

    # if 'T1' in d_keys:
    # d_keys.remove('T1')
    # d_keys.remove('T2')
    # d_keys.remove('T3')
    toggle = 'alpha1' in d_keys

    if toggle:
```

```
        d_keys.remove('alpha1')
        d_keys.remove('alpha2')

    # Inputs
    max_temp = 135

    if isinstance(meas, np.ndarray):
        if toggle:
            meas = meas[:-2]

        meas_copy = meas.tolist()
        if meas_set < 0:
            if meas_config == 'txv only':
                for idx, key in enumerate(['T1', 'T2', 'T3']):
                    del meas_copy[hex3_chen.meas_sets[meas_set].index(key)-idx]
            meas_copy = np.array(meas_copy)

            if meas_config in ('both', 'txv only'):
                actual_t1v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T1')]
                actual_t2v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T2')]
                actual_t3v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T3')]

                # Add temp backoff
                if last_run is not None:
                    if actual_t1v < 0:
                        last_run['T1V'] = last_run.get('T1V', 0) + min(0.5 * actual_t1v, -0.5)
                        actual_t1v += last_run['T1V']
                    else:
                        last_run['T1V'] = min((0, last_run.get('T1V', 0) + 0.01 * actual_t1v))

                    if actual_t2v < 0:
                        last_run['T2V'] = last_run.get('T2V', 0) + min(0.2 * actual_t2v, -0.1)
                        actual_t2v += last_run['T2V']
                    else:
                        last_run['T2V'] = min((0, last_run.get('T2V', 0) + 0.01 * actual_t2v))

                    if actual_t3v < 0:
                        last_run['T3V'] = last_run.get('T3V', 0) + min(0.10 * actual_t3v, -0.1)
                        actual_t3v += last_run['T3V']
                    else:
                        last_run['T3V'] = min((0, last_run.get('T3V', 0) + 0.005 * actual_t3v))
                # print(last_run)
                meas_copy = np.append(meas_copy, [ actual_t1v ])
                meas_copy = np.append(meas_copy, [ actual_t2v ])
                meas_copy = np.append(meas_copy, [ actual_t3v ])
            # print(meas)

        if toggle:
            # Move
            X = np.array([np.append(np.array([u_u0, u_u1]), meas_copy)])
        else:
            X = np.array([meas_copy])
    else:
        raise Exception('Not supported')
        # X = [*([u_u0, u_u1] * toggle), *[meas[k] for k in d_keys]]
        # if meas_set < 0:
        # X.append(max_temp - meas['T1'])
        # X.append(max_temp - meas['T2'])
        # X.append(max_temp - meas['T3'])
        # # X = np.append(meas, [hex3_chen.Ti_max - meas['T2']])
        # # X = np.append(meas, [hex3_chen.Ti_max - meas['T3']])
        # # print(meas)
        # X = np.array([X], dtype=np.float)
    return X


def disturbance_to_array(u, d, d_order):
    # 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['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    X = [*([u_u0, u_u1] * toggle), *[d[k] for k in d_keys]]
    X = np.array([X], dtype=np.float)
    return X


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')

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

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

    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]]
        # print(x)
        # x.extend([d_row['T1v'], d_row['T2v'], d_row['T3v']])
        # print(x)
        # print('--')
        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 generate_meas_set_data(mset, u, d, meas_config):
    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_)
        assert hex3_chen.Ti_max == 135
        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(hex3_chen.meas_sets[mset]):
            res_dict[key] = result['y'][idx]

        if meas_config in ('both', 'txv only'):
            res_dict['T1v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T1')]
            res_dict['T2v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T2')]
            res_dict['T3v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T3')]

        if meas_config == 'txv only':
            for key in ['T1', 'T2', 'T3']:
                del res_dict[key]
        # print(res_dict)
        results.append(res_dict)
    return np.array(results, dtype=dict)


class InputConstrainedController:
    def __init__(self, training_database, meas_set=None, noise=False, meas_config='txv only'):
        train_data = load_data(training_database)

        if meas_set is not None:
            # hex3_chen.meas_sets[2].exted(['T1', ...])
            d_keys = hex3_chen.meas_sets[meas_set].copy()
```

```
            if meas_config == 'txv only':
                d_keys.remove('T1')
                d_keys.remove('T2')
                d_keys.remove('T3')
            if meas_config in ('txv only', 'both'):
                d_keys.extend(['T1v', 'T2v', 'T3v'])

            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'])))
            # u3 = u1
            diff = u2 - u1
            split = int(0.30 * len(u1))
            u3 = np.concatenate((u1[:split] + diff[:split] * 0.98, u1[split:] + diff[split:] * 0.5))

            disturbances_train = generate_meas_set_data(meas_set, u3, train_data['d'], meas_config)

        else:
            raise Exception('All distrubances not supported for this controller.')
            # d_keys = None
            # disturbances_train = train_data['d']
            # u3 = train_data['u_rand']

        X_train, Y_train, key_order = preprocess(train_data['u_chen'], disturbances_train, d_order=d_keys,
                                                 u_input=u3, add_noise=noise)
        # print(X_train[0])
        # print(Y_train[0])

        self.outputs = len(Y_train[0])
        self.real_cv = np.empty((0, 2), int)
        self.t_samples = len(Y_train)

        self.meas_config = meas_config
        self.meas_set = meas_set
        self.d_keys = key_order
        self.last_violation = {}

        self.gp = GPModel()
        self.gp.start()
        self.gp.train(X_train, Y_train, num_restarts=1, normalize_y=True)

    def __str__(self):
        return 'uc'

    def plot_init(self, axes):
        lines = []
        rlines = []
        for idx, ax in enumerate(axes):
            ax.set_ylabel(f'Valve {idx + 1} [-]')
            ax.set_xlabel('Time [-]')
            ax.set_ylim([0.25, 0.35])
            ax.set_xlim([0, 1])
            line, = ax.plot([], [], label='Predicted')
            lines.append(line)
            rline, = ax.plot([], [], '--', label='Optimal')
            rlines.append(rline)
            ax.legend()
        return lines, rlines

    def gen_real_cv(self, u, u_opt, d):
        self.real_cv = np.append(self.real_cv, np.array([u_opt]), axis=0)

    def plot_prediction(self, axes, lines, time, variables, real_lines):
        # print(variables)
        for idx, (ax, line) in enumerate(zip(axes, lines)):
            ylim = ax.get_ylim()
            # print(ylim)
```

```
            if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(variables[:, idx])-0.05, np.max(variables[:, idx])+0.05])
            ylim = ax.get_ylim()
            if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(self.real_cv[:, idx])-0.05, np.max(self.real_cv[:,idx])+0.05])

            ax.set_xlim([0, len(time)])
            line.set_xdata(time)
            line.set_ydata(variables[:, idx])
            real_lines[idx].set_xdata(time)
            real_lines[idx].set_ydata(self.real_cv[:, idx])

    def predict(self, u, measurement):
        # print('.-'*30)
        # print(u, measurement)
        try:
            if self.meas_set is not None:
                X = measurements_to_array(u, measurement, self.d_keys, meas_set=self.meas_set,
                                          meas_config=self.meas_config, last_run=self.last_violation)
            else:
                raise Exception('Unhandled')
                # X = disturbance_to_array(u, measurement, self.d_keys)
        except:
            self.close()
            raise
        # print(X)
        u_old = u
        alpha = 0.95
        u, u_var, _ = self.gp.test(X) # print(f'Before {u} | grad {gradients}')
        u = u[0, :]
        return alpha*u + (1. - alpha)*np.array(u_old), alpha*u + (1. - alpha)*np.array(u_old)

    def __del__(self):
        self.close()

    def close(self):
        self.gp.exit()


if __name__ == '__main__':
    # train = f'..\\gradient\\datasets\\train_gradient{500}.csv'
    # controller = GradientController(train)
    # controller.close()
    # print('Controller closed')
    pass
```

Constrained mixed control structure:

```
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel
from optimal_u_constrained.hex3_gen_u_optim import load_data


def measurements_to_array(u, meas, d_order, meas_config, meas_set):
    # Distubances
    if isinstance(meas, pd.DataFrame):
        meas = meas.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u_u0 = u['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]
```

```
# 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')

d_keys = d_order.copy()
# d_keys.remove('T1')
# d_keys.remove('T2')
# d_keys.remove('T3')
toggle = 'alpha1' in d_keys

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

# Inputs
max_temp = 135
if isinstance(meas, np.ndarray):

    if toggle:
        meas = meas[:-2]

    meas_copy = meas.tolist()
    if meas_set < 0:
        if meas_config == 'txv only':
            for idx, key in enumerate(['T1', 'T2', 'T3']):
                del meas_copy[hex3_chen.meas_sets[meas_set].index(key)-idx]
        meas_copy = np.array(meas_copy)

        if meas_config in ('both', 'txv only'):
            actual_t1v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T1')]
            actual_t2v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T2')]
            actual_t3v = max_temp - meas[hex3_chen.meas_sets[meas_set].index('T3')]

            meas_copy = np.append(meas_copy, [ actual_t1v ])
            meas_copy = np.append(meas_copy, [ actual_t2v ])
            meas_copy = np.append(meas_copy, [ actual_t3v ])
    #
    # meas_copy = meas.copy()
    # # Remove T1, T2, T3 if selected
    # if meas_config == 'txv only':
    # for idx, key in enumerate(['T1', 'T2', 'T3']):
    # del meas_copy[hex3_chen.meas_sets[meas_set].index(key) - idx]

    if toggle:
        # Move
        X = np.array([np.append(np.array([u_u0, u_u1]), meas_copy)])
    else:
        X = np.array([meas_copy])

else:
    raise Exception('Unsupported code path')
# X = [*([u_u0, u_u1] * toggle), *[meas[k] for k in d_keys]]
# if meas_set < 0:
# X.append(max_temp - meas['T1'])
# X.append(max_temp - meas['T2'])
# X.append(max_temp - meas['T3'])
# # X = np.append(meas, [hex3_chen.Ti_max - meas['T2']])
# # X = np.append(meas, [hex3_chen.Ti_max - meas['T3']])
# # print(meas)
# X = np.array([X], dtype=np.float)
return X
```

```
def disturbance_to_array(u, d, d_order):
    # 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['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    X = [*([u_u0, u_u1] * toggle), *[d[k] for k in d_keys]]
    X = np.array([X], dtype=np.float)
    return X


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')

    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.extend([d_row['T1v'], d_row['T2v'], d_row['T3v']])
        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 generate_meas_set_data(mset, u, d, meas_config):
    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(hex3_chen.meas_sets[mset]):
            res_dict[key] = result['y'][idx]

        if meas_config in ('both', 'txv only'):
            res_dict['T1v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T1')]
            res_dict['T2v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T2')]
            res_dict['T3v'] = hex3_chen.Ti_max - result['x'][hex3_chen.x_vars.index('T3')]

        if meas_config == 'txv only':
            for key in ['T1', 'T2', 'T3']:
                del res_dict[key]

        results.append(res_dict)
    return np.array(results, dtype=dict)


class InputConstrainedController:
    def __init__(self, training_database, meas_set=None, noise=False, meas_config='txv only'):
        train_data = load_data(training_database)

        if meas_set is not None:
            # hex3_chen.meas_sets[2].exted(['T1', ...])
            d_keys = hex3_chen.meas_sets[meas_set].copy()
```

```
            if meas_config == 'txv only':
                d_keys.remove('T1')
                d_keys.remove('T2')
                d_keys.remove('T3')
            if meas_config in ('txv only', 'both'):
                d_keys.extend(['T1v', 'T2v', 'T3v'])

            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'])))
            # u3 = u1
            diff = u2 - u1
            split = int(0.70 * len(u1))
            u3 = np.concatenate((u1[:split] + diff[:split] * 0.98, u1[split:] + diff[split:] * 0.5))

            disturbances_train = generate_meas_set_data(meas_set, u3, train_data['d'], meas_config=meas_config)

        else:
            raise Exception('All distrubances not supported for this controller.')
            # d_keys = None
            # disturbances_train = train_data['d']
            # u3 = train_data['u_rand']

        X_train, Y_train, key_order = preprocess(train_data['u_chen'], disturbances_train, d_order=d_keys,
                                                 u_input=u3, add_noise=noise)
        # print(X_train[0])
        # print(Y_train[0])

        self.outputs = len(Y_train[0])
        self.real_cv = np.empty((0, 2), int)
        self.t_samples = len(Y_train)

        self.meas_config = meas_config
        self.meas_set = meas_set
        self.d_keys = key_order

        self.gp = GPModel()
        self.gp.start()
        self.gp.train(X_train, Y_train, num_restarts=1, normalize_y=True)

    def __str__(self):
        return 'uc2'

    def plot_init(self, axes):
        lines = []
        rlines = []
        for idx, ax in enumerate(axes):
            ax.set_ylabel(f'Valve_{idx + 1} [-]')
            ax.set_xlabel('Time [-]')
            ax.set_ylim([0.25, 0.35])
            ax.set_xlim([0, 1])
            line, = ax.plot([], [], label='Predicted')
            lines.append(line)
            rline, = ax.plot([], [], '--', label='Optimal')
            rlines.append(rline)
            ax.legend()
        return lines, rlines

    def gen_real_cv(self, u, u_opt, d):
        self.real_cv = np.append(self.real_cv, np.array([u_opt]), axis=0)

    def plot_prediction(self, axes, lines, time, variables, real_lines):
        # print(variables)
        for idx, (ax, line) in enumerate(zip(axes, lines)):
            ylim = ax.get_ylim()
            # print(ylim)
            if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
```

```
                ax.set_ylim([np.min(variables[:, idx]) - 0.05, np.max(variables[:, idx]) + 0.05])
            ylim = ax.get_ylim()
            if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(self.real_cv[:, idx]) - 0.05, np.max(self.real_cv[:, idx]) + 0.05])

            ax.set_xlim([0, len(time)])
            line.set_xdata(time)
            line.set_ydata(variables[:, idx])
            real_lines[idx].set_xdata(time)
            real_lines[idx].set_ydata(self.real_cv[:, idx])

    def predict(self, u, measurement):
        # print('.-'*30)
        # print(u, measurement)
        try:
            if self.meas_set is not None:
                X = measurements_to_array(u, measurement, self.d_keys,
                                          meas_set=self.meas_set, meas_config=self.meas_config)
            else:
                X = disturbance_to_array(u, measurement, self.d_keys)
                # print(X)
        except:
            self.close()
            raise

        u_old = u
        alpha = 0.8
        u, u_var, _ = self.gp.test(X) # print(f'Before {u} | grad {gradients}')

        u = u[0]

        C11u, C12u, C13u = u_old[0], u_old[0], u_old[0]
        C21u, C22u, C23u, C24u = u_old[1], u_old[1], u_old[1], u_old[1]

        N2 = np.array([0.0000, 1])
        N3 = np.array([1, 0.0000])
        N4 = np.array([-0.7071, 0.7071])

        C11u = C11u - 0.004 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T1')])
        C12u = C12u + 0.002 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T3')])
        C13u = u[0] # C13u - 0.001 * N3 @ u

        C21u = C21u - 0.004 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T2')])
        C22u = C22u + 0.002 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T3')])
        C23u = u[1] # C23u - 0.001 * N4 @ u

        # print(C11u)
        # print(C12u)
        # print(C13u)
        u1 = np.max((C11u, np.min((C12u, C13u))))
        u2 = np.max((C21u, np.min((C22u, C23u))))
        u = np.array([u1, u2])

        return u, u

    def __del__(self):
        self.close()

    def close(self):
        self.gp.exit()


if __name__ == '__main__':
    # train = f'..\\gradient\\datasets\\train_gradient{500}.csv'
    # controller = GradientController(train)
    # controller.close()
    # print('Controller closed')
    pass
```

Constrained gradient control structure:

```
import numpy as np
import pandas as pd

import hex3_chen_old as hex3_chen
from g_process import GPModel
from gradient_constrained.gradient_gen_data import load_data


def measurements_to_array(u, meas, d_order, meas_set):
    # Distubances
    if isinstance(meas, pd.DataFrame):
        meas = meas.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u_u0 = u['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

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

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

    # Inputs
    if isinstance(meas, np.ndarray):
        # Remove splits (to be re-added later down)
        if toggle:
            meas = meas[:-2]

        # print(meas)
        meas_copy = meas.tolist()

        if toggle:
            # Move
            X = np.array([np.append(np.array([u_u0, u_u1]), np.array(meas_copy))])
        else:
            X = np.array([meas_copy])

    else:
        raise Exception('Unsupported code path')

    print(X)
    return X


def disturbance_to_array(u, d, d_order):
    # 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['u0']
        u_u1 = u['u1']
    else:
        u_u0 = u[0]
        u_u1 = u[1]

    # 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')

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

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

    # Inputs
    X = [*([u_u0, u_u1] * toggle), *[d[k] for k in d_keys]]
    X = np.array([X], dtype=np.float)
    return X


def preprocess(u: pd.DataFrame, d: [pd.DataFrame, np.ndarray], g: [pd.DataFrame, np.ndarray], d_order=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['u0']
        u_u1 = u['u1']
    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')

    # TODO: Just use original variables
    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
    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.extend([d_row['T1v'], d_row['T2v'], d_row['T3v']])
        X.append(x)

    X = np.array(X)

    # Optimal gradients values
    if isinstance(g, pd.DataFrame):
        g0 = g['gc0']
```

```
        g1 = g['gc1']
    else:
        raise Exception('Unhandled datatype for gradients')

    Y = np.array(list(zip(g0, g1)))

    return X, Y, d_order


def generate_meas_set_data(mset, u, d):
    d = d.to_dict('records')

    if isinstance(u, pd.DataFrame):
        u = np.array(list(zip(u['u0'], u['u1'])))

    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(hex3_chen.meas_sets[mset]):
            res_dict[key] = result['y'][idx]

        results.append(res_dict)
    return np.array(results, dtype=dict)


class GradientConstrainedController:
    def __init__(self, training_database, meas_set=None, noise=False):
        train_data = load_data(training_database)

        hex3_chen.Ti_max = 135
        # hex3.Ti_max = 135

        u = train_data['u']
        d = train_data['d']
        g = train_data['gc']

        if meas_set is not None:

            d_keys = hex3_chen.meas_sets[meas_set]
            # print('Solving measurment set data...')
            disturbances_train = generate_meas_set_data(meas_set, u, d)

        else:
            raise Exception('All distrubances not supported for this controller.')
            # d_keys = None
            # disturbances_train = train_data['d']

        X_train, Y_train, key_order = preprocess(u, disturbances_train, g, d_order=d_keys,
                                                 add_noise=noise)
        # print(Y_train)
        # print(X_train[0])
        # print(Y_train[0])
        self.outputs = len(Y_train[0])
        self.t_samples = len(Y_train)
        self.real_cv = np.empty((0, 2), float)

        self.meas_set = meas_set
        self.d_keys = key_order
        self.gp = GPModel()
        self.gp.start()
```

```
        self.gp.train(X_train, Y_train, num_restarts=0, normalize_y=True)

    def __str__(self):
        return 'gc'

    def plot_init(self, axes):
        lines = []
        rlines = []
        for idx, ax in enumerate(axes):
            ax.set_ylabel(f'Valve {idx + 1} [-]')
            ax.set_xlabel('Time [-]')
            ax.set_ylim([0.25, 0.35])
            ax.set_xlim([0, 1])
            line, = ax.plot([], [], label='Predicted')
            lines.append(line)
            rline, = ax.plot([], [], '--', label='Optimal')
            rlines.append(rline)
            ax.legend()
        return lines, rlines

    def gen_real_cv(self, u, u_opt, d):
        self.real_cv = np.append(self.real_cv, np.array([u_opt]), axis=0)

    def plot_prediction(self, axes, lines, time, variables, real_lines):
        for idx, (ax, line) in enumerate(zip(axes, lines)):
            ylim = ax.get_ylim()
            # print(ylim)
            if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(variables[:, idx])-0.05, np.max(variables[:, idx])+0.05])
            ylim = ax.get_ylim()
            if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
                ax.set_ylim([np.min(self.real_cv[:, idx])-0.05, np.max(self.real_cv[:, idx])+0.05])

            # Fix for auto-plotting
            if self.meas_set == 1 and idx == 0:
                ax.set_ylim([ylim[0], 0.38])
            if self.meas_set == 3 and idx == 0:
                ax.set_ylim([ylim[0], 0.38])
            if self.meas_set == 4 and idx == 0:
                ax.set_ylim([ylim[0], 0.38])
            ax.set_xlim([0, len(time)])
            line.set_xdata(time)
            line.set_ydata(variables[:, idx])
            real_lines[idx].set_xdata(time)
            real_lines[idx].set_ydata(self.real_cv[:, idx])
    # def gen_real_cv(self, u, u_opt, d):
    # real_g = hex3_chen.grad(u, d)
    # assert real_g['success']
    # # print(real_g)
    #
    # self.real_cv = np.append(self.real_cv, np.array([real_g['grad']]), axis=0)
    # # print(self.real_cv)
    #
    # def plot_init(self, axes):
    # lines = []
    # rlines = []
    # for idx, ax in enumerate(axes):
    # ax.set_ylabel(f'Gradient_{idx + 1} [-]')
    # ax.set_xlabel('Time [min]')
    # ax.set_ylim([-5, 5])
    # ax.set_xlim([0, 1])
    # line, = ax.plot([], [], label='Predicted')
    # lines.append(line)
    # rline, = ax.plot([], [], '--', label='Actual')
    # rlines.append(rline)
    # ax.legend()
    # return lines, rlines
    #
```

```
    # def plot_prediction(self, axes, lines, time, variables, real_lines):
    # for idx, (ax, line) in enumerate(zip(axes, lines)):
    # ylim = ax.get_ylim()
    # # print(ylim)
    # if not (ylim[0] <= variables[-1][idx] <= ylim[1]):
    # ax.set_ylim([np.min(variables[:, idx]-1), np.max(variables[:, idx])+1])
    # ylim = ax.get_ylim()
    # if not (ylim[0] <= self.real_cv[-1][idx] <= ylim[1]):
    # ax.set_ylim([np.min(self.real_cv[:,idx])-1, np.max(self.real_cv[:,idx])+1])
    # ax.set_xlim([0, len(time)])
    # # print(time)
    # # print(variables[:, idx])
    # line.set_xdata(time)
    # line.set_ydata(variables[:, idx])
    # real_lines[idx].set_xdata(time)
    # real_lines[idx].set_ydata(self.real_cv[:, idx])

    def predict(self, u, measurement):
        # print('.-'*30)
        # print(u, measurement)
        try:
            if self.meas_set is not None:
                X = measurements_to_array(u, measurement, self.d_keys, meas_set=self.meas_set)
            else:
                # X = disturbance_to_array(u, measurement, self.d_keys)
                raise Exception('Unhandled')
        except:
            self.close()
            raise

        gradients, grad_var, _ = self.gp.test(X) # print(f'Before {u} | grad {gradients}')
        # print(gradients)
        k = -0.0005
        gradients = gradients[0]
        C11u, C12u, C13u = u[0], u[0], u[0]
        C21u, C22u, C23u, C24u = u[1], u[1], u[1], u[1]

        N1 = np.array([0.0000, 1])
        N2 = np.array([1, 0.0000])
        N3 = np.array([-0.7071, 0.7071])

        C11u = C11u - 0.007 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T1')])
        C12u = C12u + 0.004 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T3')])
        C13u = C13u - 0.001 * N2 @ gradients

        C21u = C21u - 0.007 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T2')])
        C22u = C22u + 0.004 * (hex3_chen.Ti_max - measurement[hex3_chen.meas_sets[self.meas_set].index('T3')])
        C23u = C23u - 0.001 * N3 @ gradients
        C24u = C24u - 0.001 * N1 @ gradients
        # print(C11u)
        # print(C12u)
        # print(C13u)
        u1 = max((C11u, min((C12u, C13u))))
        u2 = np.max((C21u, np.min((C24u, np.min((C22u, C23u))))))
        u = np.array([u1, u2])
        # u = u + np.array([k * gradients[0][0], k * gradients[0][1]])
        # u = np.maximum(u, [0.05, 0.05])
        # u = np.minimum(u, [0.95, 0.95])
        return u, u #gradients

    def __del__(self):
        self.close()

    def close(self):
        self.gp.exit()


if __name__ == '__main__':
```

```
    train = f'..\\gradient\\datasets\\train_gradient{500}.csv'
    controller = GradientController(train)
    controller.close()
    print('Controller closed')
```

B.5 Simulation

There the two scripts to run the constrained and unconstrained scripts are shown. The simulation script which handles the actual iteration loop for both case is also shown here.

Unconstrained starting point:

```
import code
import sys

import matplotlib
import numpy as np
import pandas
from matplotlib.ticker import FormatStrFormatter

import hex3_chen_old as hex3_chen

matplotlib.use('Qt5agg')
import matplotlib.pyplot as plt

from simulation.predictor_gradient import GradientController
from simulation.predictor_uopt import InputController
from simulation.simulation import main


def full_run(disturb_table, dist_idx):
    runs = []
    import pandas as pd
    samples = 500
    for mode in ['g', 'u']:
        for MS in range(1, 5):
            for noise in [False, True]:
                try:
                    # GP controller
                    np.random.seed(21)
                    do_plot = True

                    if mode == 'u':
                        train_data = f'optimal_u\\datasets\\train_u_prediction{samples}.csv'
                        controller = InputController(train_data, meas_set=MS, noise=noise)
                    elif mode == 'g':
                        train_data = f'gradient\\datasets\\new_train_gradient{samples}.csv'
                        controller = GradientController(train_data, meas_set=MS, noise=noise)
                    else:
                        raise Exception(f'unknown mode {mode}')

                    result = main(MS, do_plot, controller, disturb_table, noise)
                    entry = {
                        'data': result,
                        'total loss': result['total loss'],
                        'negative loss': result['negative loss'],
                        # 'max step loss': result['max step loss'],
                        'mode': mode,
                        'MS': MS,
                        'noise': noise,
                    }
                    runs.append(entry)
                    if do_plot:

                        fig = plt.gcf()
```

```
                        fig.tight_layout()
                        for idx, ax in enumerate(fig.axes):
                            # Hide all the other axes...
                            if idx in (1, 2):
                                ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
                            ax._orig_position = ax.get_position()
                            ax.set_position([0.15, 0.11, 0.80, 0.79])
                            for axis in fig.axes:
                                if axis is not ax:
                                    axis.set_visible(False)

                            # fig.tight_layout()
                            # if idx in (1, 2):
                            # plt.savefig(f'unconstrained_plots{dist_idx}\\gradients_{samples}\\unconstrained_noise_{noise}
                                ↪ _MS{MS}_mode_{mode}_{idx+1}.eps')
                            plt.savefig(f'unconstrained_plots{dist_idx}\\valves_{samples}\\unconstrained_noise_{noise}_MS{
                                ↪ MS}_mode_{mode}_{idx+1}.eps')

                            ax.set_position(ax._orig_position)
                            for axis in fig.axes:
                                if axis is not ax:
                                    axis.set_visible(True)

                        plt.close()
                    finally:
                        try:
                            controller.close()
                        except:
                            pass

    dataframe = pd.DataFrame.from_records(runs)
    pd.set_option('display.max_rows', None)
    pd.set_option('display.max_columns', None)
    pd.set_option('display.width', None)
    # pd.set_option('display.max_colwidth', -1)
    print(dataframe.sort_values(by='MS'))
    # code.interact('Interactive mode started... ', local=dict(globals(), **locals()))


def single_run(disturb_table, dist_idx):
    # mode = 'g'
    mode = 'u'
    MS = 1
    noise = False
    samples = 500
    results = {}
    try:
        # GP controller
        np.random.seed(21)
        do_plot = True
        if mode == 'u':
            train_data = f'optimal_u\\datasets\\train_u_prediction{samples}.csv'
            controller = InputController(train_data, meas_set=MS, noise=noise)

        elif mode == 'g':
            train_data = f'gradient\\datasets\\new_train_gradient{samples}.csv'
            controller = GradientController(train_data, meas_set=MS, noise=noise)

        else:
            raise Exception(f'unknown mode {mode}')

        # total_loss, max_step_loss = main(MS, do_plot, controller, distrub_table, noise)
        try:
            result = main(MS, do_plot, controller, disturb_table, noise)
        except AssertionError:
            print(f'Mode {mode} | MS: {MS} | noise: {noise}')
            plt.show()
            raise
```

```
        entry = {
            'data': result,
            'total loss': result['total loss'],
            'negative loss': result['negative loss'],
            # 'max step loss': result['max step loss'],
            'mode': mode,
            'MS': MS,
            'noise': noise,
        }

        if do_plot:

            fig = plt.gcf()
            fig.tight_layout()
            for idx, ax in enumerate(fig.axes):
                # Hide all the other axes...
                if idx in (1, 2):
                    ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
                ax._orig_position = ax.get_position()
                ax.set_position([0.15, 0.11, 0.80, 0.79])
                for axis in fig.axes:
                    if axis is not ax:
                        axis.set_visible(False)

                # fig.tight_layout()
                # if idx in (1, 2):
                # plt.savefig(f'unconstrained_plots{dist_idx}\\gradients_{samples}\\unconstrained_noise_{noise}_MS{MS}
                    ↪ _mode_{mode}_{idx+1}.eps')

                plt.savefig(f'unconstrained_plots{dist_idx}\\diverging_unconstrained_noise_{noise}_MS{MS}_mode_{mode}_{
                    ↪ idx+1}.eps')

                # if idx in (1, 2):
                # plt.savefig(f'unconstrained_plots{dist_idx}\\only_optimal_unconstrained_noise_{noise}_MS{MS}_mode_{
                    ↪ mode}_{idx+1}.eps')

                ax.set_position(ax._orig_position)
                for axis in fig.axes:
                    if axis is not ax:
                        axis.set_visible(True)

            # plt.savefig(f'unconstrained_MS{MS}_mode_{mode}_{}.png', dpi=400)
            plt.show()

        # def get_state_step(x_step, var):
        # var_step = [step[hex3_chen.x_vars.index(var)] for step in x_step]
        # return var_step
        # try:
        # # print(entry['data']['x_step'])
        # plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T1'),label='T1')
        # plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T2'),label='T2')
        # plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T3'),label='T3')
        # plt.ylim([130, 145])
        # plt.legend()
        # plt.show()
        # except:
        # import traceback
        # traceback.print_exc()

        # plt.plot(entry['data']['t_step'], entry['data']['x_step'])
        # plt.plot(entry['data']['t_step'], entry['data']['x_step'])

    finally:
        try:
            controller.close()
        except:
            pass
```

```
    import pprint
    import pandas as pd
    dataframe = pd.DataFrame.from_records([entry])
    pd.set_option('display.max_rows', None)
    pd.set_option('display.max_columns', None)
    pd.set_option('display.width', None)
    # pd.set_option('display.max_colwidth', -1)

    def to_ltx(dataframe: pandas.DataFrame):
        pd_c = dataframe.copy()
        del pd_c['data']
        print(pd_c.to_latex())

    print(dataframe.sort_values(by='MS'))
    code.interact('Interactive mode started... ', local=dict(globals(), **locals()))


if __name__ == '__main__':
    from disturbance_tables import distrub_table2, distrub_table1
    font = {'size': 14}
    matplotlib.rc('font', **font)
    disturb_table = distrub_table1

    # for idx, dist_table in enumerate([distrub_table1, distrub_table2], start=1):
    # full_run(dist_table, idx)
    # break

    # full_run(disturb_table)
    #
    single_run(disturb_table, 1)
```

Constrained starting point:

```
import code

import matplotlib
import numpy as np
from matplotlib.ticker import FormatStrFormatter

import hex3_chen_old as hex3_chen

from simulation.simulation import main

matplotlib.use('Qt5agg')
import matplotlib.pyplot as plt


def full_run(disturb_table, dist_idx, meas_config):
    runs = []
    import pandas as pd
    hex3_chen.Ti_max = 135
    samples = 500
    if meas_config == 't only': # Run gc only for t only runs, because not needed to repeat em.
        controllers = ['uc', 'uc2', 'gc']
    else:
        controllers = ['uc', 'uc2']

    for mode in controllers:
        for MS in range(1, 5):
            MS = -MS
            for noise in [True, False]:
                try:
                    # GP controller
                    do_plot = True
                    np.random.seed(21)

                    if mode == 'uc':
```

```
        from simulation.predictor_constrained_uopt import InputConstrainedController

        # train_data = f'optimal_u_constrained\\datasets\\test_u_constrained_prediction{12500}.csv'
        train_data = f'optimal_u_constrained\\datasets\\train_u_constrained_prediction{samples}.csv'
        controller = InputConstrainedController(train_data, meas_set=MS, noise=noise, meas_config=
            ↪ meas_config)
    elif mode == 'uc2':
        from simulation.predictor_constrained_uopt_v2 import InputConstrainedController

        # train_data = f'optimal_u_constrained\\datasets\\test_u_constrained_prediction{12500}.csv'
        train_data = f'optimal_u_constrained\\datasets\\train_u_constrained_prediction{samples}.csv'
        # train_data = f'optimal_u\\datasets\\train_u_prediction{500}.csv'
        controller = InputConstrainedController(train_data, meas_set=MS, noise=noise, meas_config=
            ↪ meas_config)
    elif mode == 'gc':
        from simulation.predictor_constrained_gradient import GradientConstrainedController
        train_data = f'gradient\\datasets\\new_train_gradient{samples}.csv'
        # train_data = f'gradient_constrained\\datasets\\train_constrained_gradient{500}.csv'
        controller = GradientConstrainedController(train_data, meas_set=MS, noise=noise)
    else:
        raise Exception(f'unknown mode {mode}')

    result = main(MS, do_plot, controller, disturb_table, noise)
    entry = {
        'data': result,
        'total loss': result['total loss'],
        'negative loss': result['negative loss'],
        # 'max step loss': result['max step loss'],
        'mode': mode,
        'MS': MS,
        'noise': noise,
    }
    print(entry['MS'], entry['mode'], entry['noise'])
    runs.append(entry)

    if do_plot:

        fig = plt.gcf()
        fig.tight_layout()
        for idx, ax in enumerate(fig.axes):
            # Hide all the other axes...
            if idx in (1, 2):
                ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
            ax._orig_position = ax.get_position()
            ax.set_position([0.15, 0.11, 0.80, 0.79])
            for axis in fig.axes:
                if axis is not ax:
                    axis.set_visible(False)

            # fig.tight_layout()
            plt.savefig(f'constrained_plots{dist_idx}\\{meas_config.replace(" ", "")}\\valves_{samples}\\
                ↪ constrained_noise_{noise}_MS{MS}_mode_{mode}_{idx+1}.eps')

            ax.set_position(ax._orig_position)
            for axis in fig.axes:
                if axis is not ax:
                    axis.set_visible(True)

        plt.close()

        def get_state_step(x_step, var):
            var_step = [step[hex3_chen.x_vars.index(var)] for step in x_step]
            return var_step

        try:
            # print(entry['data']['x_step'])
            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T1'),label='T1')
            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T2'),label='T2')
```

```
                            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T3'),label='T3')
                            plt.ylabel('Temperature [C]')
                            plt.xlabel('Time [-]')
                            plt.ylim([130, 145])
                            plt.legend()
                            plt.hlines(hex3_chen.Ti_max, 0, 500, colors='black')
                            plt.savefig(f'constrained_plots{dist_idx}\\{meas_config.replace(" ", "")}\\temps_{samples}\\
                                ↪ constrained_noise_{noise}_MS{MS}_mode_{mode}_temp.eps')
                            # plt.show()
                            plt.close()
                        except:
                            import traceback
                            traceback.print_exc()

                finally:
                    try:
                        controller.close()
                    except:
                        pass

    dataframe = pd.DataFrame.from_records(runs)
    pd.set_option('display.max_rows', None)
    pd.set_option('display.max_columns', None)
    pd.set_option('display.width', None)
    # pd.set_option('display.max_colwidth', -1)
    print(dataframe.sort_values(by='MS'))
    dataframe.to_pickle(f'constrained_plots{dist_idx}\\{meas_config.replace(" ", "")}\\constrained_{samples}.pkl')

    # code.interact('Interactive mode started... ', local=dict(globals(), **locals()))


def single_run(disturb_table, dist_idx, meas_config):
    mode = 'uc'
    # mode = 'uc'
    MS = -2
    noise = True

    hex3_chen.Ti_max = 135
    samples = 2500
    np.random.seed(21)
    try:
        # GP controller
        do_plot = True
        if mode == 'uc':
            from simulation.predictor_constrained_uopt import InputConstrainedController

            # train_data = f'optimal_u_constrained\\datasets\\test_u_constrained_prediction{12500}.csv'
            train_data = f'optimal_u_constrained\\datasets\\train_u_constrained_prediction{samples}.csv'
            controller = InputConstrainedController(train_data, meas_set=MS, noise=noise, meas_config=meas_config)
        elif mode == 'uc2':
            from simulation.predictor_constrained_uopt_v2 import InputConstrainedController

            # train_data = f'optimal_u_constrained\\datasets\\test_u_constrained_prediction{12500}.csv'
            train_data = f'optimal_u_constrained\\datasets\\train_u_constrained_prediction{samples}.csv'
            # train_data = f'optimal_u\\datasets\\train_u_prediction{500}.csv'
            controller = InputConstrainedController(train_data, meas_set=MS, noise=noise, meas_config=meas_config)
        elif mode == 'gc':
            from simulation.predictor_constrained_gradient import GradientConstrainedController

            # train_data = f'gradient_constrained\\datasets\\train_constrained_gradient{500}.csv'
            train_data = f'gradient\\datasets\\new_train_gradient{samples}.csv'
            controller = GradientConstrainedController(train_data, meas_set=MS, noise=noise)
        else:
            raise Exception(f'unknown mode {mode}')

        # total_loss, max_step_loss = main(MS, do_plot, controller, distrub_table, noise)
        try:
            result = main(MS, do_plot, controller, disturb_table, noise)
```

```
        except AssertionError:
            print(f'Mode {mode} | MS: {MS} | meas_config {meas_config} | noise: {noise}')
            plt.show()

        entry = {
            'data': result,
            'Normal loss': result['total loss'],
            'negative loss': result['negative loss'],
            # 'max step loss': result['max step loss'],
            'mode': mode,
            'MS': MS,
            'noise': noise,
        }
        print(entry['MS'], entry['mode'], entry['noise'])
        if do_plot:

            fig = plt.gcf()
            fig.tight_layout()
            for idx, ax in enumerate(fig.axes):
                # Hide all the other axes...
                if idx in (1, 2):
                    ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))
                ax._orig_position = ax.get_position()
                ax.set_position([0.15, 0.11, 0.80, 0.79])
                for axis in fig.axes:
                    if axis is not ax:
                        axis.set_visible(False)

                # fig.tight_layout()
                plt.savefig(f'constrained_plots{dist_idx}\\{meas_config.replace(" ", "")}\\valves_{samples}\\
                    ↪ constrained_noise_{noise}_MS{MS}_mode_{mode}_{idx+1}.eps')

                ax.set_position(ax._orig_position)
                for axis in fig.axes:
                    if axis is not ax:
                        axis.set_visible(True)

            plt.close()

        def get_state_step(x_step, var):
            var_step = [step[hex3_chen.x_vars.index(var)] for step in x_step]
            return var_step
        try:
            # print(entry['data']['x_step'])
            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T1'),label='T1')
            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T2'),label='T2')
            plt.plot(entry['data']['t_step'], get_state_step(entry['data']['x_step'], 'T3'),label='T3')
            plt.ylabel('Temperature [C]')
            plt.xlabel('Time [-]')
            plt.ylim([130, 145])
            plt.legend()
            plt.hlines(hex3_chen.Ti_max, 0, 500, colors='black')
            plt.savefig(f'constrained_plots{dist_idx}\\{meas_config.replace(" ", "")}\\temps_{samples}\\constrained_noise_{
                ↪ noise}_MS{MS}_mode_{mode}_temp.eps')
            # plt.show()
            plt.close()
        except:
            import traceback
            traceback.print_exc()

        # plt.plot(entry['data']['t_step'], entry['data']['x_step'])
        # plt.plot(entry['data']['t_step'], entry['data']['x_step'])

    finally:
        try:
            controller.close()
        except:
            pass
```

```
    import pprint
    import pandas as pd
    dataframe = pd.DataFrame.from_records([entry])
    pd.set_option('display.max_rows', None)
    pd.set_option('display.max_columns', None)
    pd.set_option('display.width', None)
    # pd.set_option('display.max_colwidth', -1)
    result = str(dataframe.sort_values(by='MS'))

    # dataframe.to_pickle(f'constrained_plots\\constrained_noise_{noise}_MS{MS}_mode_{mode}_temp.pkl')
    # code.interact('Interactive mode started... ', local=dict(globals(), **locals()))
    return result


if __name__ == '__main__':
    from disturbance_tables import constrained_distrub_table2, distrub_table1
    font = {'size': 14}
    matplotlib.rc('font', **font)
    disturb_table = distrub_table1

    results = []
    for meas_config in ('both', 't only', 'txv only'):
        for idx, dist_table in enumerate([distrub_table1, constrained_distrub_table2], start=1):
            r = single_run(dist_table, idx, meas_config)
            results.append((meas_config, idx, r))
            #full_run(dist_table, idx, meas_config)


    for meas_config, dist_set, data in results:
        print(f'Dist set {dist_set} | meas_config {meas_config}')
        print(data)

    # single_run(disturb_table)
```

Simulation:

```
import enum
import code
import matplotlib
import numpy as np
from matplotlib.backend_bases import MouseButton

import hex3_chen_old as hex3_chen

matplotlib.use('Qt5agg')
import matplotlib.pyplot as plt


class state(enum.IntEnum):
    alpha3 = 0
    T = 1
    Tstar1 = 2
    Tstar2 = 3
    Tstar3 = 4
    The1 = 5
    The2 = 6
    The3 = 7
    Q1 = 8
    Q2 = 9
    Q3 = 10
    Qloss1 = 11
    Qloss2 = 12
    Qloss3 = 13
    T1 = 14
    T2 = 15
    T3 = 16
```

```
def apply_noise(x, MS):
    noise = np.random.normal(0, 1, x.size)
    # No noise on valves
    if MS in (4, -4):
        noise[-2:] = 0

    return x + noise


def line_intersection(line1, line2):
    xdiff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    ydiff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    def det(a, b):
        return a[0] * b[1] - a[1] * b[0]

    div = det(xdiff, ydiff)
    if div == 0:
       raise Exception('lines do not intersect')

    d = (det(*line1), det(*line2))
    x = det(d, xdiff) / div
    y = det(d, ydiff) / div
    return x, y


zoomed = False
def on_click(event):
    """Enlarge or restore the selected axis."""
    global zoomed
    ax = event.inaxes

    if ax is None:
        # Occurs when a region not in an axis is clicked...
        return
    if event.button is MouseButton.LEFT:
        if zoomed:
            return
        # On left click, zoom the selected axes
        ax._orig_position = ax.get_position()
        ax.set_position([0.1, 0.1, 0.85, 0.85])
        for axis in event.canvas.figure.axes:
            # Hide all the other axes...
            if axis is not ax:
                axis.set_visible(False)
        zoomed = True
    elif event.button is MouseButton.RIGHT:
        # On right click, restore the axes
        zoomed = False
        try:
            ax.set_position(ax._orig_position)
            for axis in event.canvas.figure.axes:
                axis.set_visible(True)
        except AttributeError:
            # If we haven't zoomed, ignore...
            pass
    else:
        # No need to re-draw the canvas if it's not a left or right click
        return
    event.canvas.draw()
    event.canvas.flush_events()


# Setup system
def main(MS, do_plot, controller, disturb_table, noise):
    N = 500 # Min
    parameters = {}
```

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

parameters['Ts'] = [0, 0] # C
parameters['h1'] = [0, 0] # kW/K
parameters['h2'] = [0, 0] # kW/K
parameters['h3'] = [0, 0] # kW/K

d = {}
for i, parname in enumerate(parameters.keys()):
    d[parname] = parameters[parname][0]

u = np.array([0.3, 0.3])

# Plotting initialization
T_step = np.array([]) # T out
T_opt_step = np.array([]) # T out optimal
t_step = np.array([]) # Min, time step
d_step = np.array([])
x_step = []
num_cv = controller.outputs
u_step = np.empty((0, num_cv))
u_opt_step = np.empty((0, num_cv))
# Controlled variables
cv_step = np.empty((0, num_cv))

if do_plot:
    fig = plt.figure()
    controller_name = str(controller)

    title = f"Controller: {controller_name} | " \
            f"MS {MS} | Noise: {noise} | Samples {controller.t_samples} "
    # file_title = title.replace('|', '').replace(' ', '_')
    fig.suptitle(title)
    fig.canvas.mpl_connect('button_press_event', on_click)
    cost_ax = fig.add_subplot(221)
    state_T_line, = cost_ax.plot(t_step, T_step, label='State')
    opt_T_line, = cost_ax.plot(t_step, T_opt_step, '--', label='Optimal')
    cost_ax.set_ylabel('Cost [C]')
    cost_ax.set_xlabel('Time [-]')
    cost_ax.set_ylim([90, 135])
    cost_ax.set_xlim([0, 1])
    cost_ax.legend()

    cv_ax2 = fig.add_subplot(222)
    cv_ax2.set_ylim([110, 155])
    cv_ax2.set_xlim([0, 1])

    cv_axes = [cv_ax2]
    if num_cv == 2:
        cv_ax3 = fig.add_subplot(224)
        cv_ax3.set_ylim([110, 155])
        cv_ax3.set_xlim([0, 1])
        cv_axes.append(cv_ax3)
    elif num_cv > 2:
        raise Exception('Unhandled amount of controlled variables, for plotting.')
```

```
    cv_lines, real_cv_lines = controller.plot_init(cv_axes)

    # Integral cost plot
    loss_ax = fig.add_subplot(223)
    int_loss_line, = loss_ax.plot([], [])
    loss_ax.set_ylim([0, 10])
    loss_ax.set_xlim([0, 1])
    loss_ax.set_ylabel('Loss [C]')
    loss_ax.set_xlabel('Time [-]')

    # fig.set_dpi(160)
    # fig.show()
max_step_loss = 0
loss = 0
neg_loss = 0
old_loss = 0
loss_step = np.array([])

# Simulation loop

# sys.stderr = open(os.devnull, "w")
def disturb_system(d, t, parameters, pertub_table):
    changes = pertub_table.get(t, None)
    if changes is None:
        return d, None

    text = None
    for param, weight in changes:
        d[param] = parameters[param][0] + weight * parameters[param][1]
        if weight == 1.0:
            text = param
    return d, text

for t in range(N):
    # introduce disturbance

    # TODO: Add convenient distrubance update function.
    # Eg. by having list with [parameter, x% off nominal, time]
    # set to 0 to go back to nominal

    # 1. Th1 set 1, -1, 0, TH2....
    # 2. All 0.5
    # 3. Benchmark run

    # Tabels controllers vs loss from test

    # Maximum deviation error

    # Plot uncertainty

    d, plot_text = disturb_system(d, t, parameters, disturb_table)
    # print(d)
    x = hex3_chen.output_meas(MS, u, d)
    if noise:
        x['y'] = apply_noise(x['y'], MS)

    assert x['success']
    # print(x)
    # print(x['x'][state.T])
    # print(len(x['x']))
    x_opt = hex3_chen.optim(d)
    assert x_opt['success']
    # print(x_opt['x'][state.T])
    # print(x['y'])
    u_opt = x_opt['u']
    # if x['y'][hex3_chen.meas_sets[MS].index('T1')] > hex3_chen.Ti_max:
    # print('Constraint violation')
```

```
        # Loss calculation
        # Only positive loss
        if t != 0:
            old_loss += ((x_opt['x'][state.T] - x['T']) + (T_opt_step[-1] - T_step[-1])) / 2
            # Cases:
            # 0. Constantly at the optimal
            # 1b. Lines not crossing, positive loss
            # 1a. Lines does not cross, pure negative loss.

            # 2. Lines cross, from positive to negative loss
            # 3. Lines cross, from negative to positive loss

            current_T, last_T = x['T'], T_step[-1]
            current_opt_T, last_opt_T = x_opt['x'][state.T], T_opt_step[-1]

            # Case 0:
            if current_T == current_opt_T and last_T == last_opt_T:
                pass
            # Case 1a:
            elif current_T <= current_opt_T and last_T <= last_opt_T:
                ls = ((current_opt_T - current_T) + (last_opt_T - last_T)) * 1. / 2
                assert ls >= 0
                loss += ls
            # Case 1b:
            elif current_T > current_opt_T and last_T > last_opt_T:
                ns = ((current_opt_T - current_T) + (last_opt_T - last_T)) * 1. / 2
                assert ns <= 0
                neg_loss += ns
            # Case 2: Lines cross, from positive to negative loss
            elif current_T >= current_opt_T and last_T < last_opt_T:
                T_line = ((t-1, last_T), (t, current_T))
                T_opt_line = ((t-1, last_opt_T), (t, current_opt_T))
                t_cross, T_cross = line_intersection(T_line, T_opt_line)
                ls = (0 + (last_opt_T - last_T)) * (t_cross - (t-1)) / 2
                ns = ((current_opt_T - current_T) + 0) * (t - t_cross) / 2

                try:
                    assert ls >= 0
                    assert ns <= 0
                except AssertionError:
                    print(f'Current T: {current_T} | OPT {current_opt_T}')
                    print(f'Last T: {last_T} | OPT {last_opt_T}')
                    print(f'ns: {current_T} | OPT {current_opt_T}')

                loss += ls
                neg_loss += ns

            # Case 3: Lines cross, from negative to positive loss
            elif current_T < current_opt_T and last_T >= last_opt_T:
                T_line = ((t - 1, last_T), (t, current_T))
                T_opt_line = ((t - 1, last_opt_T), (t, current_opt_T))
                t_cross, T_cross = line_intersection(T_line, T_opt_line)

                ns = (0 + (last_opt_T - last_T)) * (t_cross - (t - 1)) / 2
                ls = ((current_opt_T - current_T) + 0) * (t - t_cross) / 2
                try:
                    assert ls >= 0
                    assert ns <= 0
                except AssertionError:
                    print(f'Current T: {current_T} | OPT {current_opt_T}')
                    print(f'Last T: {last_T} | OPT {last_opt_T}')
                    print(f'ns: {current_T} | OPT {current_opt_T}')

                loss += ls
                neg_loss += ns
            else:
                raise Exception('Unknown line behaviour')
            # if x_opt['x'][state.T] - x['T'] <= 0:
```

```
        # loss += 0
        # elif x_opt['x'][state.T]:
        # pass
        # else:
        # loss += ((x_opt['x'][state.T] - x['T']) + (T_opt_step[-1] - T_step[-1])) / 2
        # max_step_loss = max(abs(x_opt['x'][state.T] - x['T']), max_step_loss)
        # if (x_opt['x'][state.T] - x['T']) < 0:
        # print('Too hot')

    u, cv = controller.predict(u, x['y'])
    # print(f'u {u}, meas {MS}')

    # u = u_opt
    # print(cv)
    # print(f'New u: {u} | new cv: {cv}')

    # Updating control
    x_step.append(x['x'])
    cv_step = np.append(cv_step, np.array([cv]), axis=0)
    d_step = np.append(d_step, np.array([d]), axis=0)
    u_step = np.append(u_step, np.array([u]), axis=0)
    u_opt_step = np.append(u_opt_step, np.array([u_opt]), axis=0)

    T_step = np.append(T_step, x['T'])
    T_opt_step = np.append(T_opt_step, x_opt['x'][state.T])
    t_step = np.append(t_step, t)
    loss_step = np.append(loss_step, loss)
    # print(cv_step)
    # Plotting

    if do_plot:

        cost_ax.set_xlim([0, len(t_step)])
        cost_ax.set_ylim([min(T_step)-2, max(T_step)+2])
        state_T_line.set_xdata(t_step)
        state_T_line.set_ydata(T_step)

        opt_T_line.set_xdata(t_step)
        opt_T_line.set_ydata(T_opt_step)

        controller.gen_real_cv(u, x_opt['u'], d)
        controller.plot_prediction(cv_axes, cv_lines, t_step, cv_step, real_cv_lines)

        if plot_text is not None:
            for idx, cv_ax in enumerate(cv_axes):

                if cv_step[-1][idx] > controller.real_cv[-1][idx]:
                    if cv_step[-1][idx]-cv_step[-2][idx] <= 0:
                        if cv_step[-1][idx] > controller.real_cv[-1][idx]:
                            step = controller.real_cv
                        else:
                            step = cv_step

                    else:
                        step = cv_step
                else:
                    if controller.real_cv[-1][idx]-controller.real_cv[-2][idx] <= 0:
                        if cv_step[-1][idx] > controller.real_cv[-1][idx]:
                            step = controller.real_cv
                        else:
                            step = cv_step
                    else:
                        step = controller.real_cv
                # if cv_step[-1][idx]-cv_step[-2][idx] >= 0:
                # step = cv_step if cv_step[-1][idx] > controller.real_cv[-1][idx] else controller.real_cv
                # else:
                # step = cv_step if cv_step[-1][idx] < controller.real_cv[-1][idx] else controller.real_cv
                cv_ax.text(t_step[-1], step[-1][idx]+0.008*np.sign(step[-1][idx]-step[-2][idx]), plot_text)
```

```
            loss_ax.set_xlim([0, len(t_step)])
            loss_ax.set_ylim([0, loss_step[-1] + 1])
            int_loss_line.set_xdata(t_step)
            int_loss_line.set_ydata(loss_step)

            fig.canvas.draw()
            fig.canvas.flush_events()
    print('Final loss:', loss, 'Max step loss', max_step_loss, 'Negative loss', neg_loss)
    print(f'old loss: {old_loss}, sum loss + neg loss: {loss + neg_loss}')
    controller.close()
    # if do_plot:
    # plt.show()

    results = {
        'total loss': loss,
        'negative loss': neg_loss,
        'max step loss': max_step_loss,
        'cv_step': cv_step,
        'x_step': x_step,
        'T_step': T_step,
        'T_opt_step': T_opt_step,
        't_step': t_step,
        'u_step': u_step,
        'd_step': d_step,
        'u_opt_step': u_opt_step,
        'dist_table': disturb_table,
    }

    return results


if __name__ == '__main__':
    from disturbance_tables import distrub_table2, distrub_table1

    disturb_table = distrub_table1

    # full_run(disturb_table)
```

B.6 Disturbance set implementation.

Script for generating the disturbance sets.

```
from optimal_u.hex3_gen_u_optim import load_data

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

# parameters['Ts'] = [0, 0] # C
# parameters['h1'] = [0, 0] # kW/K
# parameters['h2'] = [0, 0] # kW/K
# parameters['h3'] = [0, 0] # kW/K

distrub_table1 = {}

for idx, variable in enumerate(parameters.keys()):
```

```
    distrub_table1[40*idx+10] = [(variable, 1.)]
    distrub_table1[40*idx+20] = [(variable, -1.)]
    distrub_table1[40*idx+30] = [(variable, 0.)]


distrub_table2 = {}

#
# keys = list(parameters.keys())
# rev_keys = keys.copy()
# rev_keys.reverse()
#
# for idx in range(500):
# distrub_table2[idx] = [(keys[int(str(hash(idx))[-2:]) % len(keys)], 0.2*(idx % 5)),
# (rev_keys[int(str(hash(idx+3))[-2:]) % len(keys)], -0.2*(idx % 5))]

train_data = f'optimal_u\\datasets\\test_u_prediction{2500}.csv'
distrubance_keys = list(parameters.keys())
data = load_data(train_data)

for idx in range(500):
    nearest = int(idx/10)*10
    step = idx%10
    next = int((idx+10)/10)*10

    current_d = data['d'].iloc[nearest].to_numpy()
    next_d = data['d'].iloc[next].to_numpy()

    interpolated_d = current_d * (1 - step/10) + next_d * (step / 10)

    # print(current_d)
    # print(interpolated_d)
    # print(next_d)

    key_ampltiude = [(k, (v - parameters[k][0])/parameters[k][1]) for k,v in zip(distrubance_keys, interpolated_d)]

    distrub_table2[idx] = key_ampltiude

constrained_distrub_table2 = {}
train_data = f'optimal_u_constrained\\datasets\\train_u_constrained_prediction{2500}.csv'
distrubance_keys = list(parameters.keys())
data = load_data(train_data)

for idx in range(500):
    nearest = int(idx/10)*10
    step = idx%10
    next = int((idx+10)/10)*10

    current_d = data['d'].iloc[nearest].to_numpy()
    next_d = data['d'].iloc[next].to_numpy()

    interpolated_d = current_d * (1 - step/10) + next_d * (step / 10)

    # print(current_d)
    # print(interpolated_d)
    # print(next_d)

    key_ampltiude = [(k, (v - parameters[k][0])/parameters[k][1]) for k,v in zip(distrubance_keys, interpolated_d)]

    constrained_distrub_table2[idx] = key_ampltiude
```

Thomas Edvardsen

Optimization of heat exchanger networks using Gaussian process regression

NTNU
Norwegian University of
Science and Technology