CONFIDENTIAL

Dynamic Programming based basicity control of an experimental smelting furnace prototype

G. Vitanov

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

MSCCONFIDENTIAL

Dynamic Programming based basicity control of an experimental smelting furnace prototype

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

G. Vitanov

December 8, 2022

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by Tata Steel IJmuiden BV. Their cooperation is hereby gratefully acknowledged.

Copyright © Delft Center for Systems and Control (DCSC)
All rights reserved.

Abstract

This thesis discusses the chemical composition (basicity) control problem of HIsarna, an experimental iron furnace which operates with 30% less CO_2 emissions than its traditional blast furnace counterparts. The control challenge is keeping the basicity of the plant in a narrow operating region. A mass balance model of the plant was constructed - as it is common practice in the literature for traditional blast furnaces - which we combined with a parameter search method to find the most optimal model parameters from data. Next a stochastic system model of the plant was derived using the prediction errors of the plant on a new dataset. The novelty of our work is the chosen dynamic programming controller approach that we used for controller synthesis that enables optimal control of the plant with respect to the known model error distribution. A continuous Markov Decision Process based infinite horizon controller was devised by using Value Iteration to find a fixed point in our value function space with respect to the Bellman operator: our static value function. We used multilinear interpolation and a state and inputs grid to define our value function for our continuous state space. The controller map was derived from the static value function and we used multilinear interpolation again in order to obtain a continuous controller. We validate our controller by simulating the controller performance on our stochastic system model and evaluating it versus the recorded operator runs according to our running cost function defined in the Value Iteration. In summary the resulting controller outperforms the operator on average and in the worst case has comparable performance to the operator from 1000 simulation runs. Improving the controller performance further would be possible by using a more accurate system model or using a different grid parameterization than the one we used for computational efficiency reasons.

Contents

Acknowledgements

I would like to thank my supervisors for their assistance during the writing of this thesis: Prof. Tamás Keviczky, dr. Peyman Mohajerin Esfahani, and Ir. Gyula Félix Max.

Delft, University of Technology G. Vitanov
December 8, 2022

"Maybe the journey isn't so much about becoming anything. Maybe it's about un-becoming everything that isn't really you, so you can be who you were meant to be in the first place"

— *Paul Coelho*

Chapter 1

Introduction

This thesis concerns the control of an industrial process, namely the chemical composition control of HIsarna: an experimental smelting furnace.

The structure of the Thesis is the following: In this chapter (1) we introduce our control problem that we will be solved. In chapter 2 we outline current solutions for furnace process control and the history of how those solutions came to be. In chapter 3 we examine possible modelling approaches to model the state of HIsarana, and create our system model based on the most promising approach. Next chapter 4 details the data preprocessing that was necessary for this project, and we follow with a study of system identification techniques (chapter 5) and numerical optimization methods (chapter 6). Next in chapter 7 we do a parameter search for the unknown parameters of our model of HIsarna using methods we found suitable from the previous two chapters. Next we evaluate the performance of the resulting system model in chapter 8, and follow with a study of dynamic programming in chapter 9. Finally in chapter 10 we synthesise our controller using dynamic programming and evaluate its performance. We close by summarizing our findings is chapter 11.

In order to understand the control problem first we have to understand the workings of the plant. The HIsarna pilot plant is an experimental smelting furnace that is the first of its kind in the world.

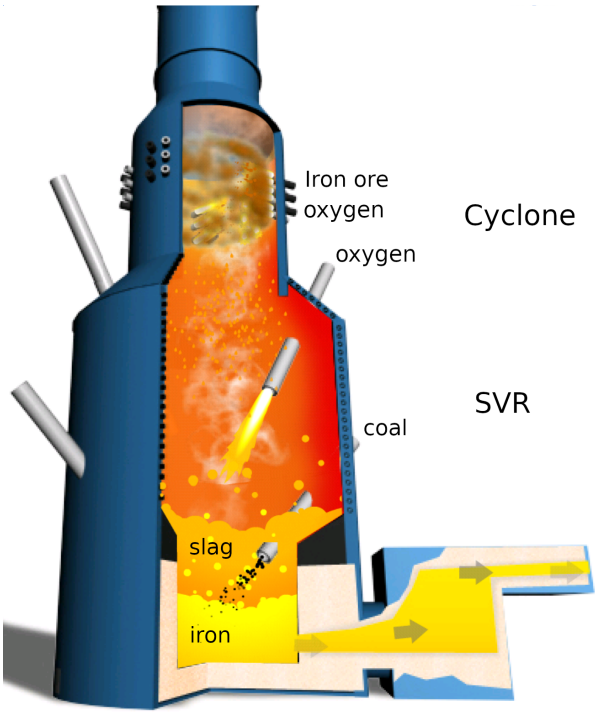

Figure 1-1: HIsarna plant overview

HIsarna is different from all other smelters in the regard that porous iron ore dust and coal dust can be directly injected into the smelter. A normal smelter requires a coal pellet plant that creates larger sized ore and coal chunks so that during smelting oxygen can flow through the coal pellets from below. HIsarna sidesteps this issue by using a different smelter geometry. Figure 1-1 shows the overview of the HIsarna plant.

The iron ore dust is injected in the top of the plant with oxygen. This top part of the plant is called the cyclone. The pre-reduction of the iron ore takes place in the cyclone. The injected iron ore (Fe_2O_3) reacts with the carbon monoxide (CO) and procures pre-reduced iron (FeO) and carbon dioxide (CO_2). The excess carbon monoxide (CO) also reacts with the injected oxygen (O_2) in the cyclone, and produces carbon dioxide (CO_2). Here oxygen is injected in to eliminate any excess carbon monoxide escaping into the atmosphere. In summary the following chemical processes take part in the cyclone:

$$\begin{aligned} Fe_2O_3 + CO &\rightarrow 2\,FeO + CO_2, \\ 2\,CO + O_2 &\rightarrow 2\,CO_2. \end{aligned} \tag{1-1}$$

Then the molten pre-reduced iron drips down the walls of the cyclone into the lower part of the plant called the SVR. In the upper part of the SVR additional oxygen is injected. The heating process of the furnace takes place here, by burning carbon monoxide (CO) with oxygen (O_2) and producing carbon dioxide (CO_2):

$$2\,CO + O_2 \rightarrow 2\,CO_2. \tag{1-2}$$

In the lower part of the SVR the materials accumulate according to density, on the very bottom the completely reduced liquid iron then on top of that the slag. The slag is a liquid material at operating temperatures which consists of different materials present in the iron ore and coal besides iron (Fe) and carbon (C). The bottom of the SVR is where the liquid iron is tapped using an overflowing dam. If the melted iron level in the furnace is sufficiently high the dam starts overflowing and the liquid iron is extracted from the furnace continuously as it is being produced. Above the liquid iron level lies the slagtap apparatus. This is used to extract slag from the furnace since slag is continuously generated and in order for the furnace to work optimally we need to be in a certain slag mass range. We also inject material into the furnace in this lower region of the SVR: this is where the coal is injected into the slag mixture. The coal (C) reduces the pre-reduced iron (FeO) completely in this region, producing hot metal (Fe) and carbon monoxide (CO). Some of the injected coal (C) is also burned in this region with oxygen (O_2) producing heat and carbon monoxide (CO),

$$\begin{aligned} &\mathrm{FeO} + \mathrm{C} \rightarrow \mathrm{Fe} + \mathrm{CO}, \\ &2\,\mathrm{C} + \mathrm{O_2} \rightarrow 2\,\mathrm{CO}. \end{aligned} \tag{1-3}$$

The slag plays an important role in the smelting process. It consist mainly of silica dioxide and calcium oxide, and also of different impurities and metal alloys other than iron and it is made from materials from the coal and ore mixture. The slag is the heat transfer medium between the upper part of the SVR - the heating zone - and the hot iron. To achieve better heat flow, and better mixing of iron and coal, the coal is injected at a very high pressure into the slag at an angle. This makes sure the coal is sufficiently mixed into the pre-reduced iron and it creates a slag fountain where droplets of slag fly around in the upper part of the SVR soaking up the heat generated by burning the coal. To achieve a good slag fountain it is important to keep the viscosity of the slag at a certain value. A good indication of slag viscosity is the basicity of the slag - the so called B_2 value - which describes a mass ratio of component materials of the slag:

$$\mathrm{B_2} = \frac{\mathrm{CaO_{mass}}}{\mathrm{SiO_{2,mass}}}. \tag{1-4}$$

Our control objective will be to keep this slag basicity value in a desired range, but ideally at a desired constant value. Our control action to do so will be a lime injection lance (hollow rod used to inject material into the furnace) in the SVR. The ore and coal mixes injected into the plant mostly contain sand (silica dioxide) and by adding lime (mostly calcium) we can affect the basicity of the slag, and steer the plant state to our desired basicity value.

The challenges of this problem are the following: The concentrations of the ore and coal mixtures can change batch by batch or even during one plant run. Thus we always have to adjust the injected lime amount to the current materials that we are using. Usually during normal operations the operators constantly have to monitor the slag concentration and make adjustments to the injected lime amount to keep the plant in the operating region. This is a very tiresome, and complex task that the operators do not achieve perfectly every time, this is why we would like to automate this process. What makes this task especially hard is that the operating region of the plant is very narrow and if we go above a certain basicity value

slag foaming can occur which can cause damage to the plant.

In summary our control goal is (i) to keep the basicity of the slag in the furnace in a safe region, (ii) preferably at a specific value by controlling the amount of lime injected into the plant.

Chapter 2

Furnace Process Control

This section will provide an overview of furnace control methods from the literature. While HIsarna is a one of a kind smelter it has the same chemical processes inside as any other smelter. This prompts us to inspect the literature to find any similar process control problems.

This section will consist of the following subsections:

- Historic developments,
- Control objectives,
- Possible modelling frameworks,
- Conclusions.

We will detail each topic and summarize the findings of my literature survey. We will also see that we can find different types of furnaces in the literature. The most common types we will be looking at are the following:

- Electric Arc Furnace,
- Blast Furnace,
- Ladle furnace.

All furnace automation tasks are revolving around controlling the amount of input materials put into the furnaces and the heating of the furnaces. The corresponding control objective might also be to achieve a certain chemical composition in the furnace, but this goal in the end also boils down to efficiently controlling the input materials of the furnace. First, lets take a look at the developments in furnace control technology that enabled us to control more efficiently the furnaces, or automate parts of these processes.

2-1 Historic developments

This section will mainly use the outline of historic developments from [1]. Initially all furnaces were just operated by a skilled operator who knew what input mixtures to use from experience, or from indicators of the hot metal. The main problem with developing a control structure for the furnaces was that estimating the state of the furnace is increasingly difficult. Direct measurement of the inside of the furnace, and hot metal is impossible even today due to the extreme heat inside the furnace. The first big advance in furnace control technology came from the development of canalization techniques of the liquid product streams of the furnace. This enables us to measure the chemical compositions of the hot metal and slag, and thus determine if any adjustment was needed to steer the furnaces into optimal operation. This enabled many advances in control technology, but still had some drawbacks. It is only possible to sample the liquid product streams of the furnace infrequently and the liquid product streams have slow dynamics. We will only be able to counteract changes on an hourly scale and also make slow adjustments.

The next advance in furnace control came form the invention of continuous gas analysis at the exhaust of the furnaces. The gas dynamic is very rapid, because the gasses have very short retention time in the furnace. This makes them a good indicator of the immediate state of the furnace. With this new data now different approaches were possible like fine tuning oxygen and coal input amounts into the furnaces. For example by measuring the CO,CO_2,O_2 concentration in the exhaust gasses we can infer if we are inputting enough oxygen to burn all the leftover coal, or if we are inputting too much oxygen, which just exits the furnace unused. These advances enabled many improvements in process control, helped keep the plant in a stable operating state and lower the costs of operation. Many publications from the next sections build on these advances and measurements.

2-2 Control objectives

There are multiple control goals in established in the Literature, a sample list follows:

- Temperature control [2] [3] [4],
- Chemical composition control [2] [5] [6],
- Avoiding slag foaming [7],
- Resource saving control [8].

In temperature control we are trying to estimate the perfect amount of coal input into the plant in order to achieve a perfect thermal balance in the plant. This is very important since the thermal variations of the smelting process greatly influence the resulting hot metal properties as described in [3]. [2]'s main objective is to reduce the impurities of the hot metal, by temperature control.

During chemical composition control we would like to achieve a certain chemical composition of our mixture. [6] details the process of accomplishing precise alloy composition control of the hot metal, [5] mainly concerns keeping the chemical composition of the hot metal (HM) mixture constant even for varying input mixtures.

An interesting goal is presented in [7], which is the necessity to avoid slag foaming during operation. This is in truth a chemical composition control problem, we highlight this separately since this also our goal with HIsarna. Slag foaming occurs when the basicity of the slag becomes too high, and this phenomenon can damage the furnace, so it is very important to completely avoid it during operation. Fortunately, by controlling the chemical composition of the slag in the furnace it is possible to completely avoid slag foaming during plant operation.

While efficient temperature control might have the side effect of reducing coal consumption, thus reducing operating cost, [8] instead focuses entirely on reducing all furnace operational costs. This means optimal oxygen, ore, alloy, coal consumption. We would like to reduce all these while still maintaining a HM quality that is requested of the plant.

We have seen multiple different main control objectives exist for achieving optimal operation of the furnaces, but in fact they all want to achieve the same thing: Stable operation of the plant, even for varying operation conditions and input mixtures. If we have stable operation then we have low coal consumption, we have a stable thermal control, and our resource allocation is optimal. This way we can also avoid slag foaming. The most important inputs of the furnaces that we can control are the

- ore amount,
- coal amount,
- oxygen,
- lime,
- additional additives.

Most controller architectures from literature concentrate on controlling one or two input types from this list. For example thermal control architectures focus on controlling the coal input and in case of the Electric Arc Furnaces the heating of the furnace. Some control architectures aim to control the oxygen in order to reduce the number of impurities found in the produced hot metal [2]. The ore amount although a very important input of the plant is usually not controlled, only set to a certain set point. This is because usually we just want a continuous hot metal production of a fixed level, and the required ore input set-point can be calculated for this easily from the chemical composition of our ore mix. Small deviations in the hot metal production are usually not significant, thus further controlling this variable is not desirable.

2-3 Possible modelling frameworks

The current most common furnace control technique is to use manual regulatory control by human operators. This is a proven strategy which works well in case of an experienced operator. Still, using human operators can introduce human errors, and have a certain limit on precision in control. It also forces plant operators to rely heavily on a small niche of expert human operators. An important advantage of automating part of the furnace operation is that it enables the operation of the plant with operators with much less experience or expertise. Lets take a look at how this automation can be achieved.

When trying to automate any process an important question to answer would be if we have access to a model of the process of not. This can influence our choice to build a model based controller or a model free controller. In this section we will go through the advantages and disadvantages of both categories.

2-3-1 Model based controllers

When we use a model based controller, we create a system model that can predict the behaviour of our system given correct measurements and inputs. With the use of this model we derive what the optimal control inputs should be when operating the system with the controller.

In order to build a model based controller we need clear understanding of the input-state-output relationships of our process. This poses a very hard obstacle when trying to model a blast furnace, since our theoretical understanding of the processes present inside the furnace during operation is still limited. This forces us to use higher level models of the furnace which only focus on basic chemical compositions, or mass balance relationships. An interesting new line of research is directed toward modelling heat, material flow, and chemical dynamics inside the furnace but these efforts are yet to produce proven results. Because our main goal is to create a control structure that will actually be implemented in the industry, we chose to disregard this new line of research, and instead focus on some thoroughly tested higher level modelling techniques.

This choice is also backed up by two more considerations from our part: Firstly, the more complex model we would like to use, the harder the model parameter identification task gets. With a large number of parameters to tune the parameter search gets increasingly difficult. In case of a non-convex optimization problem, which can often result from trying to fit nonlinear models to a known dataset, heuristic based methods like decision trees have to be used for the model parameter search. This in turn results in no guarantees of global model optimality, and model complexity usually also increases the time required to make predictions with a model. This is also an important consideration since low-complexity models lend themselves nicely to optimal control techniques, such as Dynamic Programming, where the best controller can be found numerically in a reasonable amount of time.

Since we are now familiar with the precision vs model complexity trade off of the task of modelling processes, we should also focus on the advantages of using a model based controller: We achieve better understandability of our results, and possibly more accurate results in case we have an accurate model. If we know the type of dynamics our system possesses, then using a grey-box model with system identification can produce very accurate predictions about the states of the plant. For example [6] shows that they used a mass balance model for the plant, and they were able to derive a precise model with calculated mass efficiencies of the plant.

Studies like [2] and [3] show that the most straight forward way to implement a controller for a process in a smelter usually involves an open loop control structure with a chemical model of the plant.

2-3-2 Model free controllers

A model free controller in contrast with the model based controllers, does not use model predictions to find the best control input. The controller only uses the available signals (reference inputs, output, system measurements) in order to come up with the control inputs.

The advantages of a model free controller are that it can greatly improve our workflow in case we do not know what kind of model we should use for process modelling, or in case the process is highly nonlinear and parameter estimation becomes increasingly difficult or computationally demanding. We have two choices then: Firstly, can use a hybrid approach, where we use a general function approximator as in [8] in order to learn the process model completely from data and then use a model based controller with our learned model. Secondly, we may say that we are not interested in the process model, we just want to learn with the general function approximator the required control inputs directly from the plant output as in [2]. Our decision on which method we choose will also depend on the fact that for the second option we need to know the desired control actions for each plant output. In case we do not have this information, which is often the case with furnace process control, then we need to use the hybrid approach.

The general function approximators used nowadays usually are some kind of Neural Networks. [8] show that we can create an approximate system model, with accurate predictions, that we can use later on in controller synthesis without any knowledge on the system dynamics. This study used a Kohennen Neural Network for process modelling, and used an Model Predictive Control (MPC) controller. They showed this control structure was able to significantly reduce process variation in the furnace by controlling the heating, natural gas and oxygen inputs. By using a hybrid approach we can combine the strengths on Neural Networks, with the fine control MPC provides over the plant state. With MPC we can create specific constraints on our inputs, future plant states, and more. This amount of fine control is not possible with the direct controller synthesis approach.

For direct controller synthesis we have different methods to choose from. One is to use train a neural network to have correct control actions in general operating conditions, and trust

the network to generalize well into unseen scenarios in the future. This method requires that we know the desired control actions at least for part of the operating region. This might be a large roadblock for many applications, but sometimes it is feasible. [2] used a chemical composition measurements of the hot metal to determine how much oxygen should be lanced into the furnace at each instant, and thus they could calculate the optimal input sequence. Then they trained an Artificial Neural Network model to predict the control actions, and they were able to achieve satisfactory mean percentage errors for their application, and also cost savings in operation.

Of course model free approaches do not require Neural Networks, we can also simply opt to use PID controllers, or fuzzy logic. Article [4] examines exactly this two options in the temperature control of an Electric Arc Furnace. It was found that the PID controller architecture was not able to achieve satisfactory temperature control results in the blast furnace setting. The high inertia characteristics of the process, and the nonlinear thermodynamics of the furnace make PID a suboptimal candidate for this setting. An improvement of the PID architecture can be to use a cascaded PID design. But this also performs suboptimally, because of the nonlinear process and the time varying and time delayed properties of the furnace. A satisfactory solution was finally found by using a fuzzy logic based intelligent controller in the outer loop, and a PID controller in the inner loop of the cascaded architecture.

2-4 Conclusions

We have seen many approaches for furnace process control. After weighing in the benefits and drawbacks of model based versus model free controllers the use of a model based mass balance approach for modelling HIsarna was selected. This is because in order to model our selected control target: the slag basicity inside the furnace, the SiO_2 and CaO masses inside the furnace have to be known. The mass balance model of the furnace will produce this exact information in the form of predictions, and by calculating the mass efficiencies of the certain inputs we can gain additional insight into the plant process.

It is also important to note many publications found that in order to have good results with any control approach it is important to have reliable measurement data, and also tightly control the control variables in the process. For example a common problem at smelters is keeping the chemical composition of the ore mix constant. If our ore mixture has unknown time varying properties then either controller approach will perform suboptimally. Varying ore mixtures can also be used in both model structures with more or less ease in case we do know the variation of the chemical compositions of the plant inputs. Additionally controlling the input flows of the plant is also crucial from a process control perspective. Injecting the exact required amounts can pose challenges, but this is also a point where probably most production plants can make improvements.

With keeping this in mind we can also summarize that the publications agree that there is significant gain in automating furnace processes. Possible gains are reducing human intervention, human error, freeing up workforce, achieving more stable production, better material

control, and savings in coal consumption. With technologies such as HIsara that have continuous hot metal production and more automation we are moving toward a fully automated furnace that the operator just has to switch on, set the desired set points, and the production algorithms produce the desired results.

Chapter 3

Modelling the plant

We already established we will need a mass model of the silica dioxide and calcium oxide content of the slag inside the furnace in order to gain insightful predictions on the plant state. Now we have to choose between the three possible model structures: white-box, grep-box and black-box models based on how much information we possess about the plant dynamics.

3-1 Model Types

We differentiate between three large sets of model structures as detailed by [9] depending on how much information we have about the process to model:

3-1-1 White box models

We use a white box model when we build the model completely from first principles, physical knowledge and we know all the model parameter values. This type of model does not require any kind of optimization, we just need to define all parameters and equations based on our knowledge of the system. In case we do have this knowledge this is the best model structure to use. Since we know the true value of parameters we do not need to find these values. Our model will take the following form then,

$$\hat{y} = f(x, \theta^*), \tag{3-1}$$

where

- $x \in \mathbb{X} \subseteq \mathbb{R}^n$ - is the measured inputs of the plant,
- $\theta^* \in \Theta \subseteq \mathbb{R}^o$ - is the true parametrization of the dynamics equations,
- $\hat{y} \in \mathbb{Y} \subseteq \mathbb{R}^m$ - is the predicted model output,
- $f : \mathbb{X} \times \Theta \to \mathbb{Y}$ - is the known function of system dynamics.

3-1-2 Grey box models

Grey-box models are a great choice when we do not know the exact system dynamics, but we have enough insights to come up with the parametric system equations. In this case we need to use a parameter search method in order to find the optimal parameters of our model. The parameter search for these models is usually fast, and there are a lot of well developed solutions already in the literature. For more information about grey box identification see [10],

$$\hat{y} = f(x, \theta), \tag{3-2}$$

where

- $x \in \mathbb{X} \subseteq \mathbb{R}^n$ - is the measured inputs of the plant,
- $\theta \in \Theta \subseteq \mathbb{R}^o$ - is the parametrization of the system dynamics,
- $\hat{y} \in \mathbb{Y} \subseteq \mathbb{R}^p$ - is the predicted model output,
- $f : \mathbb{X} \times \Theta \to \mathbb{Y}$ - is the known function of system dynamics.

3-1-3 Black box models

In case we do not have enough information about the system dynamics, or we are only interested in input output relationships we can use a black-box model. For a general review of black-box models see [9]. A black-box model is a general function approximator. There are different types of black box models, but in summary they all contain a family of general functions as described by [9]. During the parameter search of the black-box model we are trying to decide what combination of general functions to use to describe the input-output relationships in our data, and simultaneously we are also trying to find the best parametrization of these general functions, which we will call basis functions.

Essentially we are searching for the best mapping form the input space to the output space. In the nonlinear case, this task is usually split into two subtasks: First we create a mapping from the observable data to a so called regression vector. This vector contains all the information we can measure or calculate about our system. Next we create a nonlinear mapping from the regression vector to the output space.

3-2 Model building

As the model types section summarized well, in case we do have knowledge of the plant dynamics, it is usually advantageous to incorporate it into our system model. We know the first principle workings of our plant and we would be able to build a white-box system model with data of the chemical compositions of the input materials. Despite this we choose to use a grey-box model and a parameter search method in order to retain model flexibility and achieve a better accuracy by fitting our data directly to our ore mixture compositions. This

choice is supported by the fact that the mixing process of the ore mix is not precise and ore mix composition can vary from plant run to plant run or batch to batch. This means it is better to use data to find out model parameters than precomputed values. Figure 3-1 shows the steps of the model synthesis process concisely.

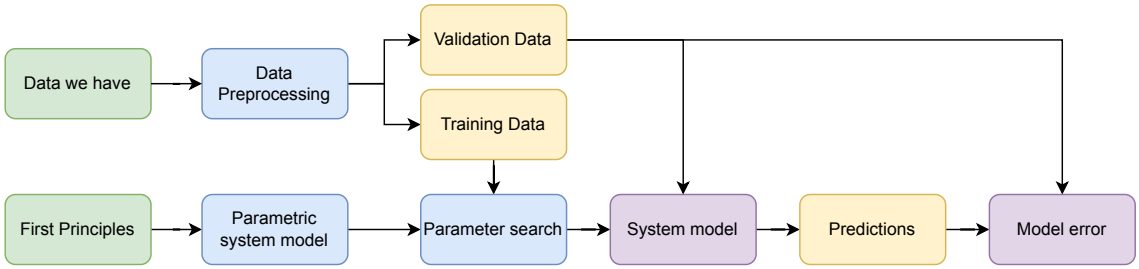

Figure 3-1: Model building process flowchart

From Figure 3-1 the green coloring shows that we have plant run data and knowledge of the chemical process in the plant as our starting points. From this data we need to create our training, validation data, and predictions of the plant behaviour, shown in yellow. To arrive at these quantities we need to do some calculations, and define processes such as the data preprocessing step, the parametric system model and the parameter search, all shown in blue. Finally if we have calculated all yellow quantities and defined all blue processes we can obtain our system model and model error distribution, shown in purple.

The individual steps of this process will be explained in separate sections of this documents, Figure 3-1 was included here to provide a brief overview of what to expect next. To understand what data we need for the model based controller synthesis, first lets look at our system model that we will use that was constructed from first principles of the plant dynamics. We decided to use a mass balance model as a result of our literature survey since it will be able to make predictions about the how our control variables change in time. We used a discrete time model because we can only measure the slag composition - the state of our plant - when slag tapping which usually only happens once in every two or three hours. This means we only have measurements of our system states at discrete time steps at irregular intervals, which meant a discrete time model was the most suitable for our control purposes. A general mass balance model has the following structure:

$$\text{next state} = \text{current state} + \text{mass efficiency matrix} * \text{inputs} - \text{outputs} \tag{3-3}$$

In equation form this will be the following:

$$x(k+1) = Ax(k) + Bu(k) - D(x(k))s_{\text{tap}}(k), \tag{3-4}$$

where,

- $x(k) \in \mathbb{X} \subseteq \mathbb{R}^3$ - is the current state vector (at time t_k, right before slagtap k),

- $x(k+1) \in \mathbb{X} \subseteq \mathbb{R}^3$ - is the predicted next state vector of the plant (at time t_{k+1}, right before slagtap $k+1$),
- $u(k) \in \mathbb{U} \subseteq \mathbb{R}^3$ - is the plant inputs vector, see explanation at (3-5),
- $s_{tap}(k) \in \mathbb{S}_{tap} \subseteq \mathbb{R}$ - is the tapped slag amount if there was any between t_k and t_{k+1} (The amount of slag tapped at slagtap k),
- $A \in \mathbb{R}^{3\times3}$ - is the mass retention matrix
- $B \in \mathbb{R}^{3\times3}$ - is the mass conversion efficiency matrix
- $D(x(k)) \in \mathbb{X} \to \mathbb{R}^3$ - is a vector function which describes how much mass is extracted (removed) from our system states at a slagtap k.

(3-4) is a nonlinear equation since the $D(x(k))$ vector is a nonlinear state dependent vector.

Our plant state ($x(k)$) and the plant inputs ($u(k)$) are the following:

$$\begin{aligned} x(k) &= \begin{bmatrix} x_0(k) \\ x_1(k) \\ x_2(k) \end{bmatrix} = \begin{bmatrix} \mathrm{Slag_m}(k) \\ \mathrm{SiO_{2,m}}(k) \\ \mathrm{CaO_m}(k) \end{bmatrix} \text{ mass in the plant at } t_k, \\ u(k) &= \begin{bmatrix} \mathrm{ore_m}(k) \\ \mathrm{coal_m}(k) \\ \mathrm{lime_m}(k) \end{bmatrix} \text{ input amount (mass) from time } t_k \text{ till } t_{k+1}, \end{aligned} \tag{3-5}$$

and we will control the lime input amount in order to achieve our control objective. We should not use the other inputs to stabilize the basicity. The $\mathrm{SiO_{2,m}}$ and $\mathrm{CaO_m}$ mass was selected as model states since they are needed to calculate our control target - the basicity - but also because they directly influence how much control action we need to apply in order to correct the basicity of the plant. The $\mathrm{Slag_m}$ mass was incorporated as a system state in order to help express the $D(x(k))$ vector. For the inputs we used the set of input variables (ore coal and lime input mnass) that significantly influence our system states.

Next question is what time step ($t_{k+1} - t_k$) do we want to use for our model? Since we can only measure the furnace state at slag tapping instances, we will use discrete time steps with varying duration to describe the system dynamics. For example we can have one measurement at 12:00, 14:00, 15:45, 18:10 and we need to be able to model all state transitions with different time frames.

Next we discuss the model assumptions we made when determining the value of the A, B matrices and the $D(x(k))$ vector function.

- If we leave the plant without inputs ($u = 0$) the amount of slag and its concentration will not change in the plant, this means the mass retention matrix will be the identity matrix. ($A = I$).
- When we are slag tapping we are tapping homogeneous slag, this means we can calculate the $D(x(k))$ vector for each slag tap as 3-6
- Time delay from input to state is insignificant.
- System is in steady state during operation, no significant periodic fluctuations exist (without u fluctuations).

List 3.1: Model assumptions derived from first principles.

From these assumptions it follows that:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad D(x(k)) = \begin{bmatrix} 1 \\ \frac{\mathrm{SiO_{2m}}(k)}{\mathrm{Slag_m}(k)} \\ \frac{\mathrm{CaO_m}(k)}{\mathrm{Slag_m}(k)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1(k)/x_0(k) \\ x_2(k)/x_0(k) \end{bmatrix}. \tag{3-6}$$

The $D(x(k))$ vector function was derived as the following: when we slagtap we are removing s_{tap} amount of slag from $\mathrm{Slag_m}$ (first row is 1). The removed slag has the following amount of $\mathrm{SiO_2}$ mass: $\mathrm{SiO_{2,m}/Slag_m}s_{\mathrm{tap}}$ because $\mathrm{SiO_{2,m}/Slag_m}$ is the current $\mathrm{SiO_{2,\%}}$ concentration of the slag, and concentration times the tapped slag weight gives us the tapped $\mathrm{SiO_2}$ weight. Similarly we can derive the tapped CaO weight.

Now the only unknown model parameter is the mass conversion efficiency matrix B, which we will calculate using a parameter search method. Our plant model from (3-4) can be expressed with a parametric B matrix,

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}, \tag{3-7}$$

as,

$$\begin{aligned} x(k+1) &= \quad A \quad x(k) + \quad B \quad U(k) - \quad D(x(k)) \quad s_{\mathrm{tap}}(k) \\ x(k+1) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} U(k) - \begin{bmatrix} 1 \\ x_1(k)/x_0(k) \\ x_2(k)/x_0(k) \end{bmatrix} s_{\mathrm{tap}}(k). \end{aligned} \tag{3-8}$$

In order to run a parameter search method to find the B matrix we first need measurements of our system states, inputs, and slagtaps (for $k = 0, 1, \ldots$):

- $\text{Slag}_\text{m}(k)$ - Slag mass at t_k,
- $\text{SiO}_{2,\text{m}}(k)$ - SiO_2 mass at t_k,
- $\text{CaO}_\text{m}(k)$ - CaO mass at t_k,
- $\text{ore}_\text{m}(k)$ - Ore input amounts (actual injected) between t_k and t_{k+1},
- $\text{coal}_\text{m}(k)$ - Coal input amounts between t_k and t_{k+1},
- $\text{lime}_\text{m}(k)$ - Lime input amounts between t_k and t_{k+1},
- $s_\text{tap}(k)$ - Slagtap (mass) amounts.

List 3.2: The data needed for parameter search

Unfortunately we do not have measurements of all these variables, so we will need to do some data preprocessing in order to arrive at the required data (List 3.2).

Chapter 4

Data Preprocessing

Currently the we cannot find all necessary data in one database only, thus we will have to combine two databases in order to obtain the data from List 3.2: The *Historian Data Consumption* database and the *Slagtap Data* database.

The *Historian Data Consumption* contains the following data:

- $\mathrm{SiO}_{2\%}(k), \mathrm{CaO}_{\%}(k)$ - Slag SiO_2 and CaO concentration measurement for each slagtap (indexed by k),
- $\mathrm{ore}_{\Delta\mathrm{m}}(i), \mathrm{coal}_{\Delta\mathrm{m}}(i), \mathrm{lime}_{\Delta\mathrm{m}}(i)$ - Injected ore, coal, and lime mass flow rate (per hour) measured every minute (indexed with i because it has different time frame),

List 4.1: Datasets contained in the Historian Data Consumption database.

while the *Slagtap Data* contains the:

- $\mathrm{Slag}_{\mathrm{inv}}(k)$ - Slag inventory mass estimates by HCM (furnace control system) before each slagtap,
- $\mathrm{Slag}_{\mathrm{make}}(k)$ - The amount of slag made between slagtaps at time t_k and t_{k+1},
- $\mathrm{Slag}_{\mathrm{tap}}(k)$ - Tapped slag mass per slagtap.

List 4.2: Datasets contained in the Slagtap Data database.

We used our own naming convention for labelling the separate data arrays to improve workflow and also because in the TATA systems we encountered 3 types of different notation for the same data so we needed some system to rename these to the same. The renaming dictionary used here is *dict_H2S*. The Slagtap Dataset labels were left unchanged. Algorithm A.1 shows the algorithm used to achieve the reformatting of the data.

We also have the problem that the Slagtap Data is in UTC time format while the Historian Data Consumption is not. This means we have to shift the time labels of the Slagtap data 2 hours to achieve Netherlands local time. This was achieved by Algorithm A.2.

Then we needed some reformatting of our datasets from List 4.1 since our data columns are different from the requirements from List 3.2:

$$\begin{aligned}
\mathrm{SiO}_{2\mathrm{m}}(k) &= \mathrm{Slag}_{\mathrm{m}}(k)\mathrm{SiO}_{2\%}(k),\\
\mathrm{CaO}_{\mathrm{m}}(k) &= \mathrm{Slag}_{\mathrm{m}}(k)\mathrm{CaO}_{\%}(k),\\
\mathrm{ore}_{\mathrm{m}}(k) &= \sum_{i=t_k}^{t_{k+1}} \frac{\mathrm{ore}_{\Delta\mathrm{m}}(i)}{60},\\
\mathrm{coal}_{\mathrm{m}}(k) &= \sum_{i=t_k}^{t_{k+1}} \frac{\mathrm{coal}_{\Delta\mathrm{m}}(i)}{60},\\
\mathrm{lime}_{\mathrm{m}}(k) &= \sum_{i=t_k}^{t_{k+1}} \frac{\mathrm{lime}_{\Delta\mathrm{m}}(i)}{60}.
\end{aligned} \tag{4-1}$$

Equation (4-1) shows that we transformed our SiO_2 and CaO concentration measurements into SiO_2 and CaO mass pseudo measurements, and we transformed the ore, coal, lime hourly mass flow measurements with a one minute time step into injected mass measurements between two slagtaps. The ore, coal and lime inputs here are calculated in the following way: Lets take the ore input as an example. We have 60 measurements of the actual ore rate in an hour, each in the metric [kg/h]. Because we only use these rates for 1 minute we have to divide them by 60 and add them up, to get the amount of actual ore mix injected in kg in a data series. This is also shown by Algorithm A.3.

On the other hand the *Slagtap Data* contains some inconsistencies: if we calculate the estimated tapped slag weight amounts from the HCM slag inventory estimates ($\mathrm{Slag}_{\mathrm{inv}}(k)$) and slag make estimates ($\mathrm{Slag}_{\mathrm{make}}(k)$), it does not equal the measured slag tap weights ($\mathrm{Slag}_{\mathrm{tap}}(k)$). Namely,

$$\mathrm{Slag}_{\mathrm{inv}}(k) - \mathrm{Slag}_{\mathrm{inv}}(k+1) + \mathrm{Slag}_{\mathrm{make}}(k) \neq \mathrm{Slag}_{\mathrm{tap}}(k) \tag{4-2}$$

This is a problem because our model from (3-4) assumes here the left hand side should equal the right hand side. Without this equivalence our model will not be accurate in its predictions. We fixed this issue by simply neglecting the tapped slag weight measurements ($\mathrm{Slag}_{\mathrm{tap}}(k)$) since according to the plant operators they were less reliable than the HCM slag inventory and slag make estimates and instead we created a new slagtap mass dataset as:

$$s_{tap}(k) = \mathrm{Slag}_{\mathrm{inv}}(k) - \mathrm{Slag}_{\mathrm{inv}}(k+1) + \mathrm{Slag}_{\mathrm{make}}(k). \tag{4-3}$$

Algorithm A.4 shows the derivation of the new slagtap mass measurement by code.

Now we have all necessary data from List 3.2. Next step is to build a parameter search framework to find the optimal values of the B matrix.

Chapter 5

System identification techniques

In this chapter we will look at an overview of the different system identification and parameter estimation approaches that could enable us to find the values of our unknown parameters of our gray-box model and create a sufficiently accurate model of HIsarna. We will do this by evaluating each approach according to their benefits and drawbacks for our application and selecting the most appropriate one.

5-1 Overview of identification methods

Depending on the model type we choose, next we have the choice of what problem formulation are we going to use:

- PEM - prediction error methods,
- SID - subspace identification methods,
- IV - instrumental variable methods.

Lets start with the prediction error methods, since those provide the most straightforward approach to parameter estimation. PEM methods are used with grey-box or black-box models.

5-1-1 Prediction error methods

As described by [11] in parameter estimation we start from an initial parametrization θ^0 of our model $f(x, u, \theta^0)$. With this parametrization we feed our model the starting states x and inputs u of our dataset ($k = 1, \ldots, n$), and calculate our predicted model outputs $\hat{y}$ as,

$$\hat{y}(k) = f(x(k), u(k), \theta^0). \tag{5-1}$$

Our task is to find the model parametrization θ^* that minimizes the error (prediction error) between the predicted model output ($\hat{y}$) and the recorded output data (y). This is done by minimizing a cost function that penalizes the difference between predicted model outputs and recorded plant outputs. For example, we can use the quadratic cost of model mismatch,

$$C(y, \theta) = \sum_{i=1}^{n} ||y_i - \hat{y}_i(\theta)||^2 . \tag{5-2}$$

By finding the model parametrization θ^* that minimizes the cost function, we find the best parameters for our model:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, C(y, \theta). \tag{5-3}$$

Substituting in the squared two norm of the model mismatch as the cost we get,

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{n} ||y_i - \hat{y}_i(\theta)||_2^2. \tag{5-4}$$

Depending on the type of cost function we use our PEM parameter estimate will coincide with the Maximum Likelihood estimate as described by [12]. Maximum Likelihood Estimation (MLE) is a topic closely related to PEM, which deals with estimation of statistical model parameters with an assumed probability density function, such that the observed data will have the highest likelihood of occurring in our statistical model with the parameter estimates obtained from MLE. For more information on the MLE estimation method see [13].

An inherent technical property of the PEM method is that the optimization step is often not solvable is closed form and we have to use numerical solutions such as the Gauss-Newton method. Unfortunately, it is also often a non-convex optimization problem, thus our gradient based numerical optimization methods will be sensitive to the initialization of θ^0. This means if we choose a random initial estimate θ^0, there is a high likelihood the Gauss-Newton optimization method will converge to a local minimum in the non-convex optimization space. This undesirable since our aim is to find the global optimum of the problem (θ^*).

In summary during the PEM parameter search our task is to find the best parameters θ^* from our parameter space Θ that minimizes our model mismatch. Depending on how we search in the parameter space we differentiate between local and global search parameter estimation methods.

5-1-2 Subspace identification

Subspace identification (SID) is used to find the state space system matrices of LTI systems. In recent years there have been developments into different applications, but we will focus on LTI systems as this is the main focus of these methods. [14] provides a great overview of different subspace algorithms. In subspace identification we are deriving the model parameters by finding subspaces of the system input-output (Hankel) data matrices with Singular Value Decomposition (SVD). From these subspaces we can reconstruct the LTI state space matrices by a linear least squares approach. A large advantage of subspace identification is

that it sidesteps the problems presented by local search methods when using a PEM problem formulation. This means subspace algorithms do not get stuck at local minima in the optimization process. On the other hand subspace algorithms provide a suboptimal solution compared to PEM in terms of numerical accuracy, see [14] Section 6.3 for more information. Subspace methods are black-box methods, since for most of them we are just defining the overall system dimension, and we have no option to incorporate previous knowledge of certain system matrices into our solution.

5-1-3 Instrumental variable methods

Instrumental variable (IV) methods on the other hand differ from PEM methods in the type of parameters they try to find. In PEM methods we are searching directly for the optimal model parameters, while in IV methods we transform our optimization problem into a pseudo linear regression form and then lowpass filter our signals (measured systems inputs, outputs, states, measurement noise) as shown by [15]. The filtering is necessary to limit noise on the derivatives of the system signals, but introduces a problem that in our filtered signals now the noise is correlated with our states and inputs. This means if we use logistic regression to calculate our model parameter estimates they will be biased. Our task is to make the noise uncorrelated to the system signals. This can be done by a transformation of our system signals: by multiplying our measurements by an instrumental variable vector. Now our goal with an optimization algorithm will be to find an instrumental variable vector that will make our noise uncorrelated to our states and inputs after transforming the signals. We effectively traded finding the best model parametrization directly to finding a good instrumental variable vector. This is advantageous to us since we have transformed our possibly non-convex search space into one with better geometry. This means our new search space resembles more closely a convex space and thus our local search methods are less likely to get stuck in local minima and are less sensitive to initialization of our initial parameter estimate than in the PEM formulation.

5-2 Conclusions

The final choice of the system identification method was to use a PEM method, since our model has relatively few parameters (nine) that we need to tune, and because our grey-box modelling approach fits a PEM framework best. PEM methods also usually provide a better estimate of the optimal system parameters than SID methods and suffer from less numerical instability or errors. Compared to IV methods PEM is easier to implement and has a more extensive literature and usage in industry.

Chapter 6

Numerical Optimization Methods

Numerical optimization methods are used when a solution to an optimization problem cannot be expressed in an explicit form, or calculated analytically. In this case we need to numerically search a parameter space in order to find the optimal solution to our problem. Most of the time we are looking for the global optimum, the best set of parameters that we can possibly find as a solution to our problem. We give each examined point from our parameter space a value by checking how good a solution that point is to our problem with a cost function we defined. There are two types of different optimization methods: local search and global search methods.

In local search methods we start from an initial estimate of parameters, and explore the vicinity of our initial estimate in the parameter space. Then we continue our search in the direction with the best results we found so far. The advantages of local search methods are that they are fast, efficient, and find the global optimum in a convex search space. The disadvantage is that they can get stuck in a local minimum for non-convex problems, and can be very sensitive to initialization. This means the results of our algorithm can greatly vary, depending on our initial parameter estimates for a non-convex problem.

This is why we use global search methods too. In global search methods we are trying to search through the whole parameter space. Of course we cannot check every point on a interval between 0 and 1 since there are infinite many points there. This means global search algorithms try to explore the whole search space, but in cases where the search space is not discrete and finite we can only do this with a certain precision. For a more extended description of search methods see [16].

6-1 Local search methods

When using a local search method we start our parameter search from an initial estimate of parameters θ^0, and use some kind of directional search method in order to determine in which

direction to search for our next θ^1 parameter estimates.

$$\theta^{i+1} = \theta^i - \mu_i R_i \nabla \hat{f}_i(\theta^i), \tag{6-1}$$

where

- μ_i is the step size,
- R_i is a matrix that modifies the search direction,
- $\nabla \hat{f}_i(\theta^i)$ is the estimate of the gradient of the cost function.

Most commonly we use the direction of the gradient or Hessian of the cost function that we are trying to minimize as our search direction. Depending on the direction metric we use, we differentiate between methods [17] such as:

- Gradient descent - $R_i = 1$,
- Gauss Newton method - $R_i = H_i^{-1}$ (where H_i is the Hessian of the cost function),
- Levenberg - Marquart method [18] [19] - $R_i = (H_i - \delta \mathbf{I})^{-1}$ where δ is an online adjusted parameter. By adjusting δ we can adjust the direction we follow in our parameter search as the following:

$$\begin{aligned} \delta \gg 1 \rightarrow R_i \approx \frac{1}{\delta}, \\ \delta \ll 1 \rightarrow R_i \approx H_i^{-1}. \end{aligned} \tag{6-2}$$

  This means by using the Levenberg-Marquart method we always choose a direction between what the gradient and the Gauss-Newton methods would have advised us ($1/\delta$ only changes the scaling, not the direction of the gradient method). Adjusting δ lets us choose how much we want to weigh the contribution of the gradient method versus the Gauss-Newton method. This lets us optimize our local search for changing cost function topology.

The fundamental problem with all local search methods is that they only guaranteed to find the global optimum of the cost function in the case the cost function is convex. Non-convex cost functions can have multiple local minima, where our local optimization methods can get stuck, producing these local minimum points as our solutions. One improvement to this problem is to carefully choose a good initialization point (initial parametrization θ^0) for our algorithms that will converge to the global optimum. Now since we do not know where the global optimum is, we can only guess for a good initialization point, thus we can improve our algorithms, but not guarantee global optimality. An other improvement of our algorithms can be to use some kind of inertia methods such as momentum and hope this way our algorithms do not get stuck in local minima and achieve faster convergence as described in [20]. This also does not guarantees global optimality although.

These problems with cost functions inspired the advent of global search methods. In comparison with local search methods these might require more resources, and also do not guarantee global optimality in all cases, but might provide an improvement over local search methods.

6-2 Global search methods

Global search methods have all in common that they all try to cover most relevant areas of the parameter space during the parameter search. The most simple algorithm is the box search.

6-2-1 Box-search

Lets say we have a model, and want to find the best model parameter with an accuracy of 1 unit, and we know the bounds of the parameter space in which to search. Lets say we have a n dimensional parameter space. In this case we just create a n dimensional parameter grid with 1 unit distance between grid points between our parameter bounds. Next we evaluate the cost function at all grid points and select the grid point with the lowest cost value as our optimal parameter vector. If our grid is sufficiently dense and or cost function is sufficiently smooth this will provide us with a near globally optimal solution with our predefined accuracy satisfied.

6-2-2 Monte-Carlo simulations

When trying to model an uncertain process we have two choices. We can replace the uncertain variables with a concrete value (e.g. most likely value or average) and look at the process outcome, or we can model the process with probability distributions. Monte Carlo simulations [21] use the later method. Essentially when trying to model a process outcome one can sample the probabilities of the random variables present in the process and calculate the process outputs thus. By the rule of large numbers, by repeating this enough times our process output distribution will be close to the true distribution in case our process model is accurate. In summary a Monte Carlo simulation uses many runs, and random distribution sampling to calculate the output probability distributions of complicated processes. This is a powerful numerical tool, because often analytical calculation of industrial processes are very complex, not feasible, or computationally much more expensive.

6-2-3 Simulated Annealing

Simulated Annealing is a metaheuristic based method for approximating the global optimum in a large search space. It is often used for discrete search spaces, where it is more important to find an approximate global optimum than to find a precise local minimum in a fixed time. This is why in these scenarios it is more preferable to exact methods such as gradient descent or branch and bound. Simulated annealing is a probabilistic exploration of the state space. As shown in [22] each iteration the algorithm starts form a certain state s and looks at a randomly chosen neighbor of the current state and chooses with a certain probability to move the current state to this next state. The probability of moving the current state is dependent on the so called temperature variable which is explained next: Simulated annealing gets its name from the metallurgical annealing process where a metal is heated then a controlled cooling phase sets the desired material properties. As explained in [23] in simulated annealing we use the same temperature and slow cooling concept as in metallurgy. The slow cooling

process in simulated annealing translates into a decrease in temperature, and thus in the probability of the algorithm accepting worse solutions than the current state. This way at the start of the algorithm when the temperature is high we accepts worse next states, and thus we can explore the state space better, but as temperatures drop we start to converge toward the global optimum with our current state. Typically the algorithm is run until a good enough state is found or the computational budget is exhausted.

6-2-4 Random initialization

An other common practice is to use a local optimization method and run it multiple times with random initializations [24]. This way even if some of the initializations converge to different local optima, we can hope that at least one initialization will converge to the global optimum position, although we get no guarantees of this happening. Of course computationally this is an expensive method running a lot of different local searches.

6-3 Conclusions

The Box search global search method was chosen as the parameter search method to be used in our PEM system identification method. While box search is only fitting for low dimensional search spaces our model has nine parameters to tune. We sidestep this issue by doing three separate box searches to find all model parameters. This is possible since our system model has a separable structure. This is explained more extensively in the next chapter.

Chapter 7

Parameter search

This chapter details how the parameter search method works for finding the optimal B matrix of our model. Our model is,

$$x(k+1) = Ax(k) + Bu(k) - D(x(k))s_{\text{tap}}(k). \tag{7-1}$$

We will use our model as a one ahead predictor. This means we only try to predict the next plant state from the current measured plant state. We can do this one ahead prediction on a time series of measurements: In order to calculate the one ahead predictions for all time samples we can use the matrix $\boldsymbol{x}_{meas}$ which has all the $x(k)$ measurements ($k \in \{0, 1, \ldots, n\}$), and similarly $\boldsymbol{u}, \boldsymbol{D}(\boldsymbol{x}_{meas}), \boldsymbol{s}_{\text{tap}}$:

$$\begin{aligned}
\boldsymbol{x}_{meas} &= \begin{bmatrix} x(0) & x(1) & \ldots & x(n) \end{bmatrix} = \begin{bmatrix} \text{Slag}_\text{m}(0) & \text{Slag}_\text{m}(1) & \ldots & \text{Slag}_\text{m}(n) \\ \text{SiO}_{2,\text{m}}(0) & \text{SiO}_{2,\text{m}}(1) & \ldots & \text{SiO}_{2,\text{m}}(n) \\ \text{CaO}_\text{m}(0) & \text{CaO}_\text{m}(1) & \ldots & \text{CaO}_\text{m}(n) \end{bmatrix}, \\
\boldsymbol{u} &= \begin{bmatrix} u(0) & u(1) & \ldots & u(n) \end{bmatrix}, \\
\boldsymbol{D}(\boldsymbol{x}_{meas}) &= \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \frac{\text{SiO}_{2\text{m}}(0)}{\text{Slag}_\text{m}(0)} & \frac{\text{SiO}_{2\text{m}}(1)}{\text{Slag}_\text{m}(1)} & \ldots & \frac{\text{SiO}_{2\text{m}}(n)}{\text{Slag}_\text{m}(n)} \\ \frac{\text{CaO}_\text{m}(0)}{\text{Slag}_\text{m}(0)} & \frac{\text{CaO}_\text{m}(1)}{\text{Slag}_\text{m}(1)} & \ldots & \frac{\text{CaO}_\text{m}(n)}{\text{Slag}_\text{m}(n)} \end{bmatrix}, \\
\boldsymbol{s}_{\text{tap}} &= \begin{bmatrix} s_{\text{tap}}(0) & s_{\text{tap}}(1) & \ldots & s_{\text{tap}}(n) \end{bmatrix}.
\end{aligned} \tag{7-2}$$

Then the predicted next state matrix is the following,

$$\boldsymbol{x}_{pred} = A\boldsymbol{x}_{meas} + B\boldsymbol{u} - \boldsymbol{D}(\boldsymbol{x}_{meas}) \otimes \boldsymbol{s}_{\text{tap}}, \tag{7-3}$$

we can rewrite (7-3) with our measurement data at hand using the following knowledge:

$$x(k) = D(x(k))\text{Slag}_\text{m}(k), \tag{7-4}$$

into,

$$\boldsymbol{x}_{pred} = \boldsymbol{D}(\boldsymbol{x}_{meas}) \otimes (\mathbf{Slag_m} - \boldsymbol{s}_{\text{tap}}) + B\boldsymbol{u}. \tag{7-5}$$

Our measured output $\boldsymbol{y}$ is the same as the state matrix $\boldsymbol{x}_{meas}$ time shifted by one step,

$$\begin{aligned} y(i) &= x_{meas}(i+1), \\ \boldsymbol{y} &= \begin{bmatrix} \text{Slag}_\text{m}(1) & \dots & \text{Slag}_\text{m}(n+1) \\ \text{SiO}_{2\text{m}}(1) & \dots & \text{SiO}_{2\text{m}}(n+1) \\ \text{CaO}_\text{m}(1) & \dots & \text{CaO}_\text{m}(n+1) \end{bmatrix}. \end{aligned} \tag{7-6}$$

Next we can calculate the error between the model predictions and our measurements as:

$$C(\boldsymbol{y}, \boldsymbol{x}_{pred}) = \sum_{i=1}^{n} (y_{i,0} - x_{pred,i,0})^2 + \sum_{i=1}^{n} (y_{i,1} - x_{pred,i,1})^2 + \sum_{i=1}^{n} (y_{i,2} - x_{pred,i,2})^2, \tag{7-7}$$

Now while we could use gradient descent and similar algorithms to find the optimal values of the B matrix, all these methods proved quite slow and computationally demanding, and do not guarantee a global optimum for non-convex optimization problems. Grid search on the other hand provides a sufficiently good estimate of the global optimum with a definable desired precision. Moreover since we can search for the optimal parametrization of the rows of the B matrix separately, we can use grid search on a 3D box three times to find the optimal parameters quickly with a desired precision. This separate search for the B matrix parameters is possible since our (7-3):

$$\begin{bmatrix} x_{pred,0}(k) \\ x_{pred,1}(k) \\ x_{pred,2}(k) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Slag}_\text{m}(k) \\ \text{SiO}_{2,\text{m}}(k) \\ \text{CaO}_\text{m}(k) \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} \text{ore}_\text{m}(k) \\ \text{coal}_\text{m}(k) \\ \text{lime}_\text{m}(k) \end{bmatrix} - \begin{bmatrix} 1 \\ \frac{\text{SiO}_{2,\text{m}}(k)}{\text{Slag}_\text{m}(k)} \\ \frac{\text{CaO}_\text{m}(k)}{\text{Slag}_\text{m}(k)} \end{bmatrix} s_{\text{tap}}(k), \tag{7-8}$$

can be separated into the following equations:

$$\begin{aligned} x_{pred,0}(k) &= \text{Slag}_\text{m}(k) + \begin{bmatrix} b_{11} & b_{12} & b_{13} \end{bmatrix} \begin{bmatrix} \text{ore}_\text{m}(k) \\ \text{coal}_\text{m}(k) \\ \text{lime}_\text{m}(k) \end{bmatrix} - s_{\text{tap}}(k), \\ x_{pred,1}(k) &= \text{SiO}_{2,\text{m}}(k) + \begin{bmatrix} b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} \text{ore}_\text{m}(k) \\ \text{coal}_\text{m}(k) \\ \text{lime}_\text{m}(k) \end{bmatrix} - \frac{\text{SiO}_{2,\text{m}}(k)}{\text{Slag}_\text{m}(k)} s_{\text{tap}}(k), \\ x_{pred,2}(k) &= \text{CaO}_\text{m}(k) + \begin{bmatrix} b_{31} & b_{32} & b_{33} \end{bmatrix} \begin{bmatrix} \text{ore}_\text{m}(k) \\ \text{coal}_\text{m}(k) \\ \text{lime}_\text{m}(k) \end{bmatrix} - \frac{\text{CaO}_\text{m}(k)}{\text{Slag}_\text{m}(k)} s_{\text{tap}}(k), \end{aligned} \tag{7-9}$$

with the following separate cost functions:

$$\begin{aligned} C(\boldsymbol{y}_0, \boldsymbol{x}_{pred,0}) &= \sum_{i=1}^{n} (y_0(i) - x_{pred,0}(i))^2, \\ C(\boldsymbol{y}_1, \boldsymbol{x}_{pred,1}) &= \sum_{i=1}^{n} (y_1(i) - x_{pred,1}(i))^2, \\ C(\boldsymbol{y}_2, \boldsymbol{x}_{pred,2}) &= \sum_{i=1}^{n} (y_2(i) - x_{pred,2})(i)^2. \end{aligned} \tag{7-10}$$

This separation is possible since we are only using the model for a one ahead prediction and thus on the right hand side of (7-9) Slag_m,$\text{SiO}_{2\text{m}}$,CaO_m are all measurements.

The gird search algorithm can be found in the appendix as Algorithm A.6 and works as the following: First we choose an area inside the search space and create a cube (3D) grid inside this area with a certain precision, this gives us a set of grid points in this area Θ. Then we test each grid point θ from our set of grid points Θ with the cost function:

$$C(\boldsymbol{y}_j, \boldsymbol{x}^{\theta}_{pred,j}) = \sum_{i=1}^{n} (y_j(i) - x^{\theta}_{pred,j}(i))^2, \tag{7-11}$$

where,

$$\begin{aligned} x^{\theta}_{pred,j}(k) &= x_{meas,j}(k) + B_j(\theta)u(k) - D_j(x_{meas}(k))s_{\text{tap}}(k), \\ B_j(\theta) &= \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 \end{bmatrix}. \end{aligned} \tag{7-12}$$

After a grid search we can easily find the best θ by finding the lowest $C(\theta)$. Now in order to find all parameters of B we have to do 3 grid searches each on a different row of B. After all unknown parameters of the B matrix have been found we end up with a system model which is optimized for making prediction one step in the future. We optimized our model for this type of one ahead prediction since during normal operation of the plant the controller will also rely on the last measurement of plant state available to it in order to minimize the expected cost of the next plant state (we will use a dynamic programming controller). Now that we have our one ahead system model we will evaluate its accuracy in the next chapter.

Chapter 8

Model evaluation

When we have our system model we can check how well it performs when predicting the next plant states. We will use 2 methods to evaluate model performance: error histograms and correlation coefficients.

8-1 Error Histograms

We can create an error histogram by using the model to create predictions about the plant state for a recorded run as in (7-5), using $\boldsymbol{y}$ from (7-6) and calculating the model error for all time steps as,

$$\boldsymbol{x}_{err} = \boldsymbol{y} - \boldsymbol{x}_{pred}. \tag{8-1}$$

We can calculate the mean vector of our dataset (bias), and the covariance matrix by,

$$\overline{x}_{err} = \mathbb{E}\left[X_{err}\right] = \frac{1}{n}\sum_{i=1}^{n} x_{err}(i), \tag{8-2}$$

$$\text{cov}(\boldsymbol{x}_{err}) = \mathbb{E}\left[\left(X_{err} - \mathbb{E}\left[X_{err}\right]\right) \cdot \left(X_{err} - \mathbb{E}\left[X_{err}\right]\right)^{\top}\right] = \frac{1}{n-1}(\boldsymbol{x}_{err} - \overline{x}_{err}) \cdot (\boldsymbol{x}_{err} - \overline{x}_{err})^{\top}. \tag{8-3}$$

Next we create an error histogram and fit a normal distribution to the histogram data since we noticed our error distribution resembles a normal distribution. We will use this assumption later on to characterise our model error distribution as a normal distribution with mean and covariance calculated by (8-3). Figure 8-1 shows the error histograms used to visualize the model prediction error in blue, the bias by the red vertical line, and the fitted normal distribution by the dashed red curve. A good model has a thin error spread, and a bias close to 0.

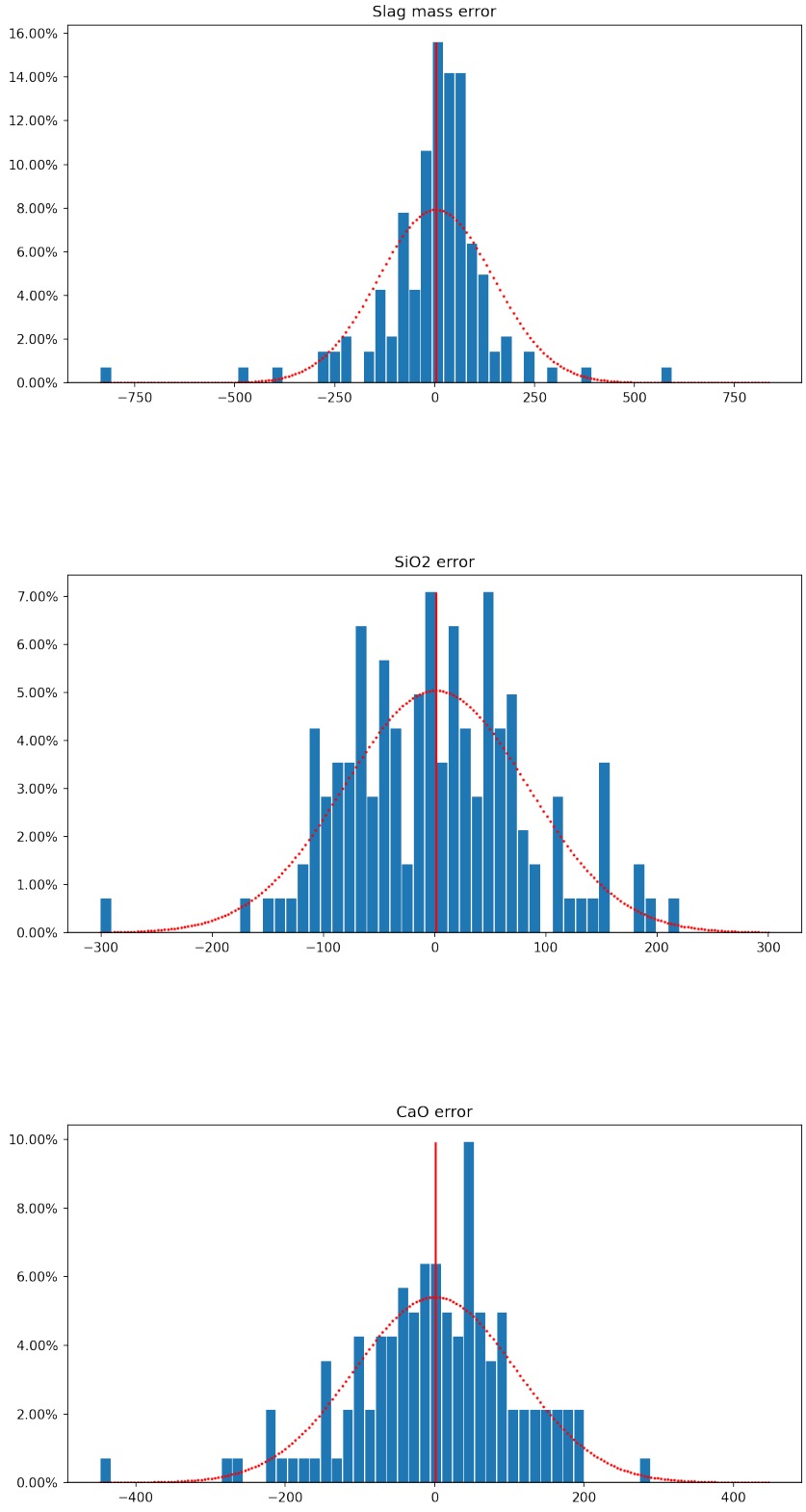

Figure 8-1: Error histograms of the model predictions on the validation dataset.

8-2 Correlation coefficients

An other important statistical method to check model performance is to calculate the correlation between our error signals and our inputs and desired outputs. We will use the Pearson correlation coefficient for this check since it can recognize linear correlation between two one dimensional datasets. We can calculate the Pearson correlation coefficient r by

$$r = \frac{\text{cov}(\boldsymbol{x}_{err,j}, \boldsymbol{s})}{\sigma(\boldsymbol{x}_{err,j})\sigma(\boldsymbol{s})}, \tag{8-4}$$

where we are interested in if $\boldsymbol{x}_{err,j}$ correlates with signal $\boldsymbol{s}$ (for example inputs or states of the plant), and $\text{cov}(a, b)$ is the covariance (scalar) between datasets (1D) a and b and σ_a is the standard deviation of the dataset a which is calculated as,

$$\sigma(\boldsymbol{x}_{err,j}) = \sqrt{\mathbb{E}\left[(X_{err,j} - \overline{X}_{err,j})^2\right]} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x}_{err,j}(i) - \overline{x}_{err,j})^2}, \tag{8-5}$$

where $\overline{x}$ is calculated according to (8-3). The resulting Pearson correlation table is shown below:

	Slag_m	$\text{SiO}_{2\text{m}}$	CaO_m	ore_m	coal_m	lime_m	s_tap
$\text{Slag}_{\text{pred,err}}$	0.04	0.04	0.05	-0.16	-0.16	-0.08	-0.02
$\text{SiO}_{2\text{pred,err}}$	0.09	0.19	0.06	-0.10	-0.13	-0.08	0.10
$\text{CaO}_{\text{pred,err}}$	0.16	0.13	0.27	-0.02	-0.02	-0.05	0.18

Table 8-1

We can see there is weak correlation between some of our inputs, measurements and our model prediction errors. We interpreted this weak correlation as a satisfactory result for model performance since our prediction errors have a zero mean (are predictions are unbiased).

There is one more metric we can look at in order to see how our model performs which is similar to the error histograms but have additional information. If we look at the error histograms from Figure 8-1 we only understand how precise our model is in predicting the different system states but we cannot see if the errors on one state correlate with the errors on an other state. To better understand this property we can look at the errors of our model plotted on a 2D plane with respect to each other as shown on Figure 8-2.

Figure 8-2: Model errors plotted

We are not interested in our slag mass prediction errors since they do not influence our control objective. This means we will only use the SiO_{2m} and CaO_m prediction errors from now on:

$$\boldsymbol{x}_{err,12} = \begin{bmatrix} \boldsymbol{x}_{err,1} \\ \boldsymbol{x}_{err,2} \end{bmatrix} \tag{8-6}$$

We already established that we presume our errors follow a normal distribution we can fit a 2D normal distribution to this dataset ($\boldsymbol{x}_{err,12}$). We can do this simply by calculating the mean and covariance of our SiO_{2m} and CaO_m errors by (8-3). If we have the mean and covariance of our 2D normal error distribution we can calculate the probability density of each point (x) on the 2D plane assuming a Gaussian distribution by (as Algorithm A.7 shows),

$$\text{pdf}(x) = \frac{1}{\sqrt{(2\pi)^2 ||\text{cov}(\boldsymbol{x}_{err,12})||}} e^{-\frac{1}{2}(x-\overline{x}_{err,12})^\top \cdot \text{cov}(\boldsymbol{x}_{err,12})^{-1} \cdot (x-\overline{x}_{err,12})}. \tag{8-7}$$

Using the pdf function from (8-7) we can plot the probability density contour lines onto Figure 8-3.

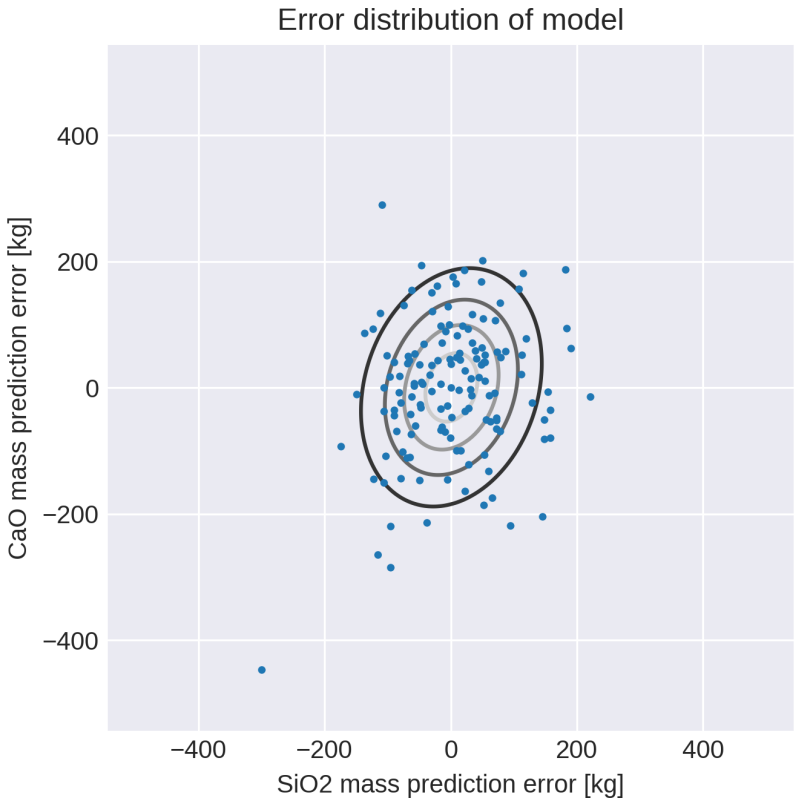

Figure 8-3: Contour lines of the probability density function, model errors inside the inner ellipses are more probable than outside

Thus on Figure 8-3 we can see that the SiO_2 and CaO prediction errors are slightly correlated since the contour lines trace out ellipses whose semi-mayor axis does not lie on the x or y axis.

The fact that we can calculate the probability density of our model errors now will enable us in the controller synthesis section to improve the expected controller performance. While now we have a continuous probability density function of our model errors, later on we will need a discretized model error set instead on a continuous one.

The discrete model error set will be selected as follows: We create a set of grid points (1000×1000) on the 2D plane of model errors, with our error distribution at the center of the grid. This grid will represent our model error distribution. Next we create the probability matrix by evaluating the probability density function (8-7) at every grid point. We normalize the probability matrix so that its elements sum up to one. Next we calculate the cumulative distribution function from the probability matrix and find the 0.99 level set of the cumulative distribution function. Then we create a rectangle grid on the model error plane which covers exactly the 0.99-th level set of the cumulative distribution function, and we evaluate the probability of every grid point with (8-7). We save the set of grid points with probability

larger than the 0.99-th level set value of the cumulative distribution function as our discrete error distribution. This means we created a discrete error set which covers the 99-th percentile of possible model errors. Figure 8-4 shows the model error points (blue), the 99-th level set (green), and the set of discretized probability density function points (red) on the model errors plane.

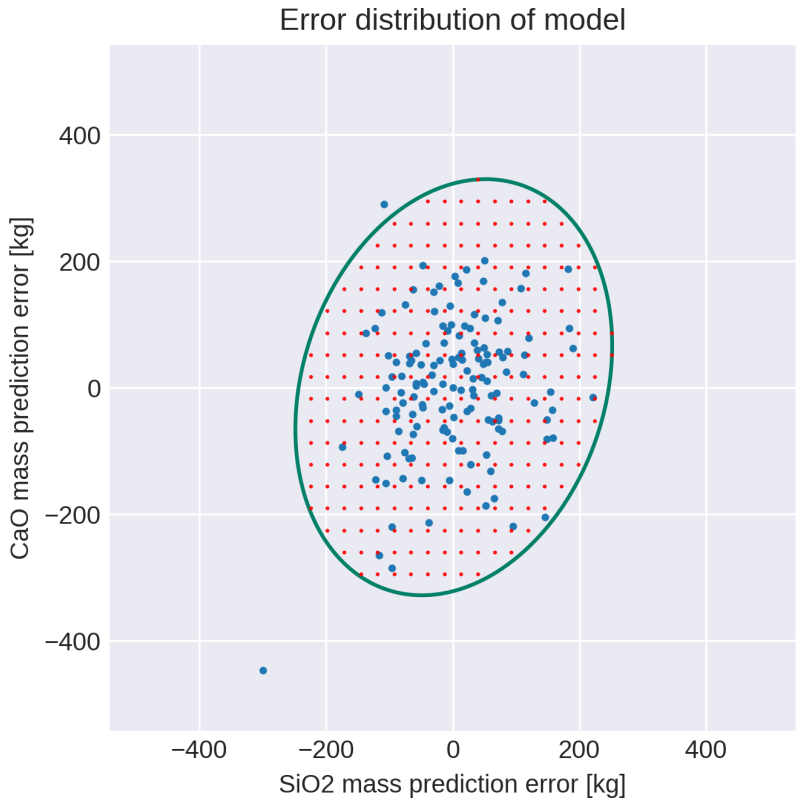

Figure 8-4: Model error points (blue), the 99-th level set (green), and the set of discretized probability density function points (red) on the model errors plane.

Algorithm A.8, A.9, and A.10 show the calculation of the discrete error set $\mathbb{W}$ by code. With the discrete error set at hand we can move on to the controller synthesis chapter. We conclude this chapter by declaring our model sufficiently accurate to use with our controller based on the mean and variance of prediction errors, and the low linear correlation of prediction errors with measurements and inputs.

Chapter 9

Dynamic Programming

We already know the plant model is not perfect and has errors on all predictions. If we want to take into account the inaccuracies of our model during controller synthesis, we can transform our model into a stochastic process model with an approximate discrete error distribution. Using a stochastic model results in a stochastic control problem where we are interested in operating the plant in the optimal operating point indefinitely. This means we have an infinite horizon stochastic control problem. This infinite dimensional optimization problem is not solvable with conventional optimization methods so we will use Dynamic Programming (DP) to partition our large optimization problem into a series of separately solvable deterministic optimization problems as shown in [25].

9-1 Introduction

Dynamic programming nowadays is a field of its own with many applications in different sectors such as economics [26] [27], ecology [28] [29], hydro power [30] [31], and scheduling [32]. Its wide use is justified by the fact that it is a very versatile tool for decision making under uncertainty.

A prerequisite of us applying dynamic programming to our control problem is that we can write our stochastic control problem as a Markov Decision Process (MDP). An MDP [33] is a stochastic process model type where the probability of transitioning into a certain next state of the system only depends on the current state and the control action we apply,

$$P_u(X_{n+1} = x_{n+1}|X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = P_u(X_{n+1} = x_{n+1}|X_n = x_n). \quad (9\text{-}1)$$

Equation (9-1) shows a Markov decision process, P_u is the transition probability of transitioning into state x_{n+1} from state x_n when applying action u. This requirement is satisfied since our discrete state space model (3-4) that we chose in chapter 3 is and MDP model.

The dynamic programming formalization is the following [34]: we are at state x and we are searching for the best control action to use u^*. We can calculate the cost of our control actions as $g(x,u)$. In order to evaluate how good a certain control action is, we not only need to know the cost of that control action $g(x,u)$, but also the value of the state that we end up in applying that control action. This means we need to construct a value function that tells us the inherent value of each state. To calculate the value function ($V(x)$) we will use the Bellman operator (Γ):

$$\Gamma(V)(x) := \min_{u \in \mathbb{U}} g(x,u) + \beta \int V(\hat{x}) p(\mathrm{d}\hat{x}|x,u), \tag{9-2}$$

where

- $g(x,u)$ - is the running cost of being in state x and applying action u,
- β - is the discount factor,
- $\mathrm{V}(\hat{x})$ - is our current estimate of the value function,
- $\int V(\hat{x}) p(\mathrm{d}\hat{x}|x,u) = \mathbb{E}_{\hat{x}}[V(\hat{x})]$ - is the expected value of the next state ($\hat{x}$) if we are in state x and apply action u.

We get our next value function estimate then as $V_1(x) = \Gamma(V_0)(x)$. An important property of the Bellman operator is that it is a β contraction mapping on the space of value functions (V). This means by repeatedly applying the Bellman operator Γ we are converging to one stationary function ($\overline{V}$) (with respect to the Bellman operator) in our value function space. We take advantage of this property of the Bellman operator in many numerical methods to calculate convergent approximate solutions to the Bellman's equation. There are many different methods to calculate the value function for an infinite horizon dynamic programming problem, but the most common one to use it the successive approximations method [27]. In the successive approximations method we use this property of the Bellman operator to arrive at an optimal estimate of the value function by applying the Bellman operator over and over again until we reach a convergent stationary value function ($\overline{V}(x)$).

$$\begin{aligned} V_1(x) &= \Gamma(V_0)(x), \\ V_2(x) &= \Gamma(V_1)(x), \\ &\vdots \\ \overline{V}(x) &:= \Gamma(\overline{V})(x), \end{aligned} \tag{9-3}$$

From the stationary value function we can the derive our controller as,

$$U_{\mathrm{con}}(x) := \operatorname*{argmin}_{u \in \mathbb{U}} g(x,u) + \beta \int \overline{V}(\hat{x}) p(\mathrm{d}\hat{x}|x,u). \tag{9-4}$$

For the first approximation of the value function for an infinite horizon dynamic programming problem we may just use any value for example zeros:

$$V_0(x) = 0. \tag{9-5}$$

Before diving deeper into the calculations, first lets see what are the different choices we can make when selecting a DP method.

We can differentiate between continuous MDP-s and discrete MDP-s, and depending on our choice we may need to use different algorithms to solve our control problem. The difference between the two is our state space. In continuous MDP-s our state space is a continuous set while in discrete MDP-s our states are discrete. This means our state transition probabilities are also continuous and discrete respectively.

9-2 Discrete MDPs

Now since our transition probabilities are a discrete set we need to redefine the Bellman (9-2) to its discrete form as described in [28],

$$\Gamma(V)(x) = \min_{u \in \mathbb{U}} g(x,u) + \beta \sum_{\hat{x} \in \mathbb{X}} V(\hat{x}) p(\hat{x}|x,u). \tag{9-6}$$

For most practical applications we can define a finite discrete state space and action space that is of interest. For a finite state space we can calculate the value function with the use of matrix algebra as,

$$\Gamma(V) = \min_{u}(g_u + \beta P_u V), \tag{9-7}$$

where

$$V := \begin{bmatrix} v_{x_1} \\ v_{x_2} \\ \vdots \\ v_{x_n} \end{bmatrix}, \qquad P_u := \begin{bmatrix} p_{x_1,x_1} & p_{x_1,x_2} & \cdots & p_{x_1,x_n} \\ p_{x_2,x_1} & p_{x_2,x_2} & \cdots & p_{x_2,x_n} \\ \vdots & \ddots & \cdots & \vdots \\ p_{x_n,x_1} & p_{x_n,x_2} & \cdots & p_{x_n,x_n} \end{bmatrix}, \qquad g_u := \begin{bmatrix} g_{x_1} \\ g_{x_2} \\ \vdots \\ g_{x_n} \end{bmatrix}, \tag{9-8}$$

where v_{x_a} is the value of the state x_a and p_{x_a,x_b} from the matrix (P_u) is the probability of transitioning from state x_a to state x_b if we applied control action u. The cost matrix is g_u where g_{x_a} is the cost of being in state x_a and applying control action u, and β is the discount factor scalar.

If we use full matrix notation P becomes a tensor (3 dimensional) with indexing $P(u, x_a, x_b)$, and the cost matrix will be a matrix (2 dimensional) as $g(u, x)$. Then we can calculate $\Gamma(V)$ as a minimization along the first matrix dimension.

While calculating (9-7) is straight forward, we will need to use a different method since in our MDP model of HIsarna our states are continuous. This means we will need to use the continuous MDP modelling framework.

9-3 Continuous MDP approaches

We have already seen how the discrete MDP approaches can calculate the Discrete Bellman (9-6) in the previous section. Our main problem to solve in the Continuous MDP approach is how to calculate the multivariate integral from the continuous Bellman equation (9-2). We have two choices here: we can either use discrete approximation, or continuous approximation.

As described by [35] during discrete approximation we transform our continuous problem into a discrete MDP. First, we select a set of discrete points form the state space ($\mathbb{X}_d$) and action space ($\mathbb{U}_d$), that are going to be our discrete states and actions. Next, we calculate the transition probabilities between our discrete states for each action applied to arrive at the state transition probability tensor ($P(u_d, x_{d_1}, x_{d_2})$). Finally we can use the discrete Bellman equation (9-6) to calculate the value functions value at our defined set of discrete state space grid points ($V(x_d)$). The resulting value function can only be evaluated at the predefined grid points of the state space $V(x_d)$.

While performing a discrete approximation of a continuous MDP can be easily done, it comes with some drawbacks: If we want a precise approximation of the true value function we will need to use dense grids for our state and action space which requires large computational power. This is where continuous approximations of the value function have many advantages, namely we can arrive at better value function estimates with less computation. The main difference between continuous approximation and discrete approximation of the value function is that in the continuous case the state space and action space can be continuous thus our value function and transition probability function will also be continuous. Numerous methods of continuous and smooth approximation can be found in [25]. Here smooth approximation denotes the fact that these methods produce value functions whose derivates exist in every point. It is important to note in continuous approximation of V we also have to select a set of grid points from the state and action space. These will be our interpolation points later on, where we fit our basis functions to the value function. In the continuous approximation methods the value function can be evaluated at any point in the state space $V(x)$, and we can decide what basis functions we would like to use to approximate V. The most common basis functions are the following:

- Multilinear interpolation,
- Chebyshev polynomials,
- Piecewise polynomials.

The multilinear interpolation is the most simple from all methods. It is a linear approximation along each dimension of the state space. In this method first we need to select a grid of the state space where we will calculate the value of V. The difference compared to the discrete MDP method is that we will be able to calculate values of $V(x)$ where $x \notin \mathbb{X}_d$ (where the state x does not lie on a grid point) using interpolation.

Other smooth approximation techniques work similarly. We parameterize our basis functions with a parameter vector θ and we select a set of grid points for the state and action space

(x_d, u_d) where we calculate $V_{k+1}(x_d)$ from a previous value function estimate $\hat{V}_k(x, \theta_k)$.

$$V_{k+1}(x_d) = \Gamma(\hat{V}_k(x_d, \theta_k)) \tag{9-9}$$

Next we try to find the best parameterization of θ such that our next estimate of the value function $\hat{V}_{k+1}(x_d, \theta)$ is closest to our points $V_{k+1}(x_d)$.

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{n} (V_{k+1}(x_{d,i}) - \hat{V}_{k+1}(x_{d,i}, \theta))^2 \tag{9-10}$$

To better understand the computational advantages and disadvantages of these different methods we have to first examine the computational complexity of dynamic programming problems.

9-4 The curse of dimensionality

The curse of dimensionality is an important concept in dynamic programming. It [36] states that when we add a dimension to the state space the number of computations required to derive the value function increases exponentially. This is a problem since often times we would like to solve stochastic optimization problems with dynamic programming with multidimensional state spaces. It affects both the discrete MDP and the continuous time MDP approaches but differently.

When using a discrete MDP approach and we increase the dimension of our state space by one, we need to use much more grid points. For example we have a 3D discrete cube grid as our state space, with 10 possible states along each dimension: 10^3 possible states altogether. If we add one more dimension to our state space the number of grid points increase to 10^4 immediately. This means we have to do an order of magnitude more computations to derive the value function.

The problem with continuous MDP-s is different. If we add one more dimension we might have to increase the number of parameters in θ in order to account for this change. Now the problem is if we increase the number of parameters in θ that greatly increases the search space of our optimization step when we are trying to fit $\hat{V}(x, \theta)$ to $V(x)$.

Essentially by increasing the dimension of the state space in the both the discrete and continuous MDP approaches we get an exponentially increasing number of grid points, and thus an exponentially increasing number of calculations to find the value function $V(x_d)$.

While the curse of dimensionality affects all MDP problems we have tools to reduce the leading constants of the exponential dependence:

- sparse matrices,

- parallel computations,
- action elimination,
- smooth approximation.

Lets walk through how these methods work. When dealing with a discrete or discretized MDP problem we have a large state transition probability matrix. This matrix is usually sparse, assuming smooth dynamics since from one state we can transition into only so many neighbouring states. We can use this property of the state transition matrix in order to speed up our value function calculations. If we have a sparse matrix we can transform it into such a structure that the matrix computations required to calculate the value function can run more efficiently, thus resulting in a speedup. This is done nowadays automatically in matrix computation libraries if they detect sufficient structure so we do not need to spend additional attention on this topic.

The next point is parallel computations. While we need to run the value function approximation steps in series, the computations of one step of the value function approximation can be very well parallelized. For example we can calculate separately,

$$\begin{aligned} \Gamma(V)(x_1) &= \min_{u \in \mathbb{U}} g(x_1, u) + \beta \int V(\hat{x}) p(\mathrm{d}\hat{x}|x_1, u) \\ \Gamma(V)(x_2) &= \min_{u \in \mathbb{U}} g(x_2, u) + \beta \int V(\hat{x}) p(\mathrm{d}\hat{x}|x_2, u). \end{aligned} \tag{9-11}$$

This can produce significant speedups if we are able to code our calculations to run on a GPU, or even just parallelize our calculations on multi core CPU. For example by running the calculations parallel on 64 cores we can achieve a speedup of approximately 50x (some data copying operations reduce the speedup from 64x).

An other method is action elimination: we do not to calculate the value of every state action pair in the value function

$$\begin{aligned} \Gamma(V)(x) = \min_{u \in \mathbb{U}_{\mathrm{red}}} & g(x, u) + \beta \int V(\hat{x}) p(\mathrm{d}\hat{x}|x, u) \\ \text{where} \qquad & \mathbb{U}_{\mathrm{red}} \subset \mathbb{U}, \end{aligned} \tag{9-12}$$

but only for a subset (U_{red}) of all actions (U). In many scenarios depending on our current state a lot of possible actions might not make sense outright. By eliminating these actions from the calculations, we can increase our algorithms speed. For certain special scenarios: if our model is linear and our cost function convex with respect to the control action then applying the Bellman operator will not change the convexity of our value function. This means we can use gradient descent to solve the minimization problem and find the optimal control action which can be much faster than testing all possible inputs u. It is also possible in this case to use a continuous action space directly instead of a discrete one, since we can find the ideal action with a certain precision with a finite number of computations with this method.

Finally the last type of speedup is achievable with smooth approximation. Lets say we have a one dimensional value function for the following example: If we know that the value function

takes the shape of a parabola then we can fit this parabola from only using three points as our state grid contrary to using a lot of grid points to approximate the parabola with multilinear interpolation. This can provide a large speedup of our calculation since the total calculation time is proportional to the size of our state grid. On the other hand we can only use this method to reduce the computational complexity of our problem if we have priori knowledge of the expected shape of the value function. Otherwise we have to use a family of general functions as our smooth functions and many grid points in order to achieve a good fit between our smooth approximation of our value function and the grid points where we calculate the actual value function value.

In summary the most promising speedup algorithms are parallel computation and action elimination. Because these methods if properly implemented can reduce the required computations and increase the speed of our calculations by orders of magnitude.

9-5 Conclusions

The conclusions of the dynamic programming literature review part are the following: Since our chosen model of HIsarna has a continuous state space, we will need to use a continuous MDP approach to calculate the value function. From the continuous MDP approaches we will use the multilinear interpolation technique since it is easy to implement and we do not have priori knowledge of the expected shape of the value function. This means we cannot reduce the number of state grid points used drastically using smooth approximation. In order to further speed up computations the following candidate solutions were deemed worthy of trial: parallel computation, and action elimination.

Chapter 10

Controller Synthesis

This section will detail the methodology used for creating the controller of the plant. We will build on the continuous MDP approach outlined in the previous chapter, explain each step in detail and highlight modifications that lead to speed up.

We selected a continuous MDP approach with multilinear interpolation of the value function. This means we have to define a grid of points where we will calculate the value of the value function and then interpolate it between the grid points. We will call these grid points the *possible starting states* ($\hat{\mathbb{X}}^0$) and the *possible inputs* ($\hat{\mathbb{U}}_f, \hat{\mathbb{U}}_c$) where $\hat{\mathbb{U}}_f$ is the grid for the fixed inputs and $\hat{\mathbb{U}}_c$ is the grid for our control input.

Figure 10-1 shows the overview of the controller synthesis process.

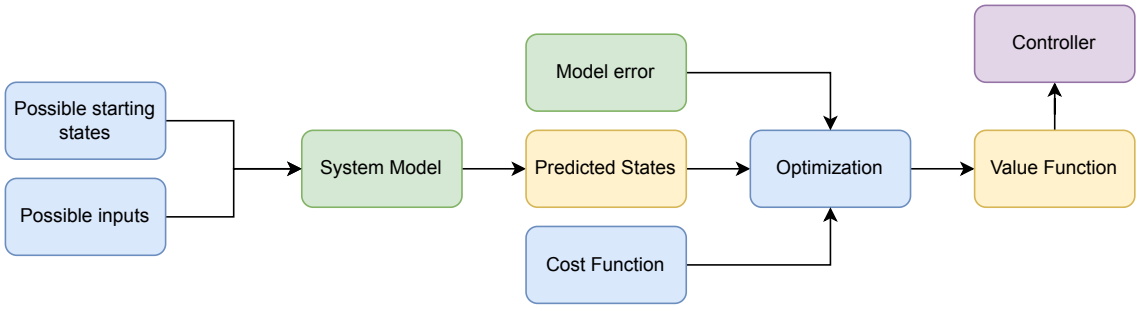

Figure 10-1: Controller synthesis process flow chart

The green boxes form Figure 10-1 show what we have already obtained in the previous chapters: the system model and model error distribution. The blue boxes contain everything we still need to define such as the possible states of our model, the possible input combinations, and the optimization method for our controller. We also need to define a metric that will tell the optimization algorithm what is the cost of each state input combination, this will be our

running cost function g. In yellow we can see the quantities we need to calculate, such as the predicted states of the plant and the value function of our controller. Purple shows us our desired end product, which is the controller table.

10-1 Introduction

We established in the previous chapter the value function of a continuous MDP problem can be calculated as (9-2), but we have to slightly modify this framework since our control action and thus our value function will not only depend on the state of the plant x but also on the fixed inputs of the plant u_f. This is necessary because we only want to use the lime input u_c of the plant in order to control the basicity, but we also have other inputs of the plant (ore and coal injected) that have a major affect on system dynamics. The modified Bellman equation is then,

$$\Gamma(V)(x, u_f) := \max_{u_c \in \hat{\mathbb{U}}_c} g(x, u_f, u_c) + \beta \int V(\tilde{x}, u_f) p(\mathrm{d}\tilde{x}|x, u_f, u_c). \tag{10-1}$$

After iteratively refining the value function by repeatedly applying the Bellman operator to (10-1) as shown by (9-3) we arrive at the stationary value function $\overline{V}(x, u_f)$. We can derive our controller from the stationary value function as,

$$U_{\text{con}}(x, u_f) := \operatorname*{argmax}_{u_c \in \hat{\mathbb{U}}_c} g(x, u_f, u_c) + \beta \int \overline{V}(\tilde{x}, u_f) p(\mathrm{d}\tilde{x}|x, u_f, u_c), \tag{10-2}$$

Our next tasks will be:

- Define starting states and inputs grids $\hat{\mathbb{X}}^0, \hat{\mathbb{U}}_f, \hat{\mathbb{U}}_c$
- Calculate predicted states X^+
- Define running cost function $g(x^0, u_f, u_c)$
- Implement multilinear interpolation to calculate $V(x, \hat{u}_f)$
- Calculate the expected value of predicted state distribution $\mathbb{E}_{\hat{x}}[V(\hat{x})] = \int V(\tilde{x}, u_f) p(\mathrm{d}\tilde{x}|x, u_f, u_c)$.

10-1-1 Starting states and inputs

To get a good approximation of the continuous value function by multilinear interpolation we have to choose our grid points carefully. We also need to make sure to cover the operating region of the plant with our grid points in order to be able to use interpolation when calculating $V(\hat{x}, u_f)$. We select our grid points based on our system states x and fixed inputs u_f, and control inputs u_c that can arise during normal operation. As we already mentioned we needed to incorporate here the fixed inputs of the plant because we do not want to optimize the plant state using them but they affect the system dynamics in major ways. This increases the required number of grid points significantly (see explanation at section 9-4). To counteract

this we will not use a grid for the Slag mass system state, instead we will only use a scalar. This will be explained below more extensively. We also assumed that slag taps do not change the system basicity in our system model, so for our starting states and inputs grid we will disregard the slag taps happening $s_{\text{tap}} = 0$. We will use equidistant grids for every variable. To protect the intellectual property of TataSteel we will also not publish the true values of all grids used during the project, but instead opt to show how the grids are constructed with different randomly chosen values. The chosen grids for the variables are as follows.

- $x_1^0 \in \hat{\mathbb{X}}_1^0 \subset [5000,\ 15000]$ with $|\hat{\mathbb{X}}_1^0| = 50$ - the SiO_2 mass state in the furnace will have an equidistant grid with 50 points starting form 5000 kg till 15000 kg. These starting and endpoints for the grid were chosen because 5000 kg is 80% of the minimal SiO_2 mass measured in the furnace in our available data and 15000 kg is 120% of the maximal measured SiO_2 mass, so this range covers the operating region and allows some lower and higher values also. We will denote with $x_{1,i}^0$ one element of this grid.
- $x_0^0 \in \hat{\mathbb{X}}_0^0(x_{1,i}^0) = x_{1,i}^0/0.4$ - The Slag mass state will not have a grid, but only a scalar value. This is done in order to reduce the number of grid points we use to reduce the required number of computations in controller synthesis. While the Slag mass does influence the system dynamics it does not influence our control objective: the plant basicity. For example if our plant has a SiO_2 mass, b CaO mass and c and d fixed inputs, then no matter what the current slag mass is we should inject e lime in order to arrive at our basicity target. Here the constant 0.4 is the average SiO_2 mass concentration of the Slag.
- $x_2^0 \in \hat{\mathbb{X}}_2^0(x_{1,i}^0) \subset [1.19x_{1,i}^0;\ 1.31x_{1,i}^0]$ where $|\hat{\mathbb{X}}_2^0| = 20$ - this means for each grid point of the SiO_2 mass grid $x_{1,i}^0$ there will be a different x_2^0 grid
- $u_1 \in \hat{\mathbb{U}}_{f_1} \subset [10000;\ 50000]$ where $|\hat{\mathbb{U}}_1| = 30$ - ore grid
- $u_2 \in \hat{\mathbb{U}}_{f_2}(u_{1,k}) \subset [u_{1,k}/2 - 8000;\ u_{1,k}/2 + 8000]$ where $|\hat{\mathbb{U}}_2| = 40$ - coal grid (on average for every 1 kg of injected coal we inject 2 kg ore).
- $u_c \in \hat{\mathbb{U}}_c \subset [0;\ 1]$ where $|\hat{\mathbb{U}}_c| = 51$ - lime grid

Table 10-1 summarizes the chosen grid parameters:

	grid starting value	grid end value	num.
$\hat{\mathbb{X}}_0^0$	$x_{1,i}^0/0.4$	$x_{1,i}^0/0.4$	1
$\hat{\mathbb{X}}_1^0$	5000	15000	50
$\hat{\mathbb{X}}_2^0$	$1.19x_{1,i}^0$	$1.31x_{1,i}^0$	20
$\hat{\mathbb{U}}_{f_1}$	10 000	50 000	30
$\hat{\mathbb{U}}_{f_2}$	$u_{1,k}/2 - 8000$	$u_{1,k}/2 + 8000$	40
$\hat{\mathbb{U}}_c$	0	1	51

Table 10-1: Grid points for the dynamic programming controller synthesis

Instead of defining static intervals for every variable as grids we let the grid points depend on each other. This is done in order to have good coverage of the operating region of the plant while using as few points as possible.

10-1-2 Predicted states

In the previous section we have defined our controller grid. This means we have defined a set of starting states $\hat{\mathbb{X}}^0$, and a set of possible inputs $\hat{\mathbb{U}}^0$. Now we will define the tensors that contain every possible state vector and input vector combination from our grids $\hat{\mathbb{X}}^0, \hat{\mathbb{U}}^0$. We will define X^0 as the tensor that has every possible $X^0_{i,j}$ combination (we will call this the tensor with every possible starting state) and U^0 as the four tensor that has every $U^0_{k,l,m}$ combination (the inputs tensor).

$$X^0_{i,j} = \begin{bmatrix} x^0_{1,i}/0.4 \\ x^0_{1,i} \\ x^0_{2,j}(x^0_{1,i}) \end{bmatrix} \in \hat{\mathbb{X}}^0, \qquad U^0_{k,l,m} = \begin{bmatrix} u_{1,k} \\ u_{2,l}(u_{1,k}) \\ u_{c,m} \end{bmatrix} \in \hat{\mathbb{U}}^0, \tag{10-3}$$

where

$$\begin{bmatrix} u_{1,k} \\ u_{2,l}(u_{1,k}) \end{bmatrix} \in \hat{\mathbb{U}}_f, \qquad u_{c,m} \in \hat{\mathbb{U}}_c \tag{10-4}$$

where i, j, k, l, and m are the grid indices that go from 0 till the max grid sizes. The i-th row and j-th column of X^0 has the starting state vector $X^0_{i,j}$. With the possible starting states tensor X^0 and possible inputs tensor U^0 at hand we can calculate the possible next states using the system model. The set of possible next states denote all the points in the state space where we can end up at using any combination of possible starting states and possible inputs.

$$\begin{aligned} X^+ &= \text{model}(X^0, U^0) \\ X^+_{i,j,k,l,m} &= X^0_{i,j} + BU^0_{k,l,m} \end{aligned} \tag{10-5}$$

Here we are presuming that no slag tap happens $s_{\text{tap}} = 0$. This is in order to reduce the complexity of the dynamic programming synthesis. This assumption is also reasonable since our control objective only depends on the basicity which we assumed does not change during a slag tap. We denote in (10-5) X^+ the six tensor that contains all possible next states, starting from any possible starting state X^0 and using any possible input combination U^0. Algorithm A.11 shows how X^+ was calculated in python. Figure 10-2 shows the X^0 grid points in red and the X^+ points in blue on the SiO_2 - CaO state space.

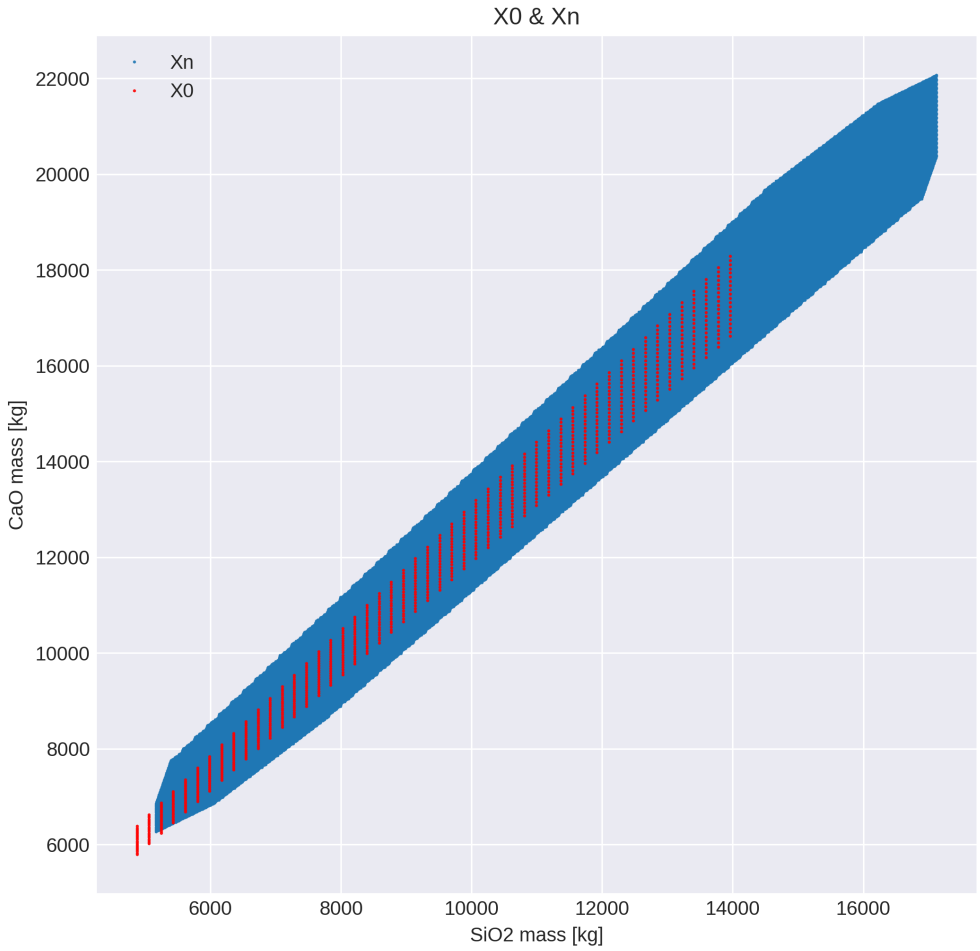

Figure 10-2: The chosen $X^0_{i,j}$ (red) grid points and $X^+_{i,j,k,l,m}$ (blue) next state points.

We can see from Figure 10-2 that the X^+ states cover a larger area than the X^0 states especially toward the high SiO_2 mass region. This is undesirable since when we calculate $V(X^+_{i,j,k,l,m})$ we will need to use interpolation or extrapolation since we only know the concrete value of the value function at $V(X^0_{i,j})$ and we use multilinear interpolation to calculate it elsewhere. This is necessary since many points from our predicted state set are not in our starting states set ($X^+_{i,j,k,l,m} \notin \hat{\mathbb{X}}^0$ for many i, j, k, l, m).

This means when we calculate all $V(X^+_{i,j,k,l,m})$ for some i, j, k, l, m combinations we will need to use extrapolation since this particular $X^+_{i,j,k,l,m}$ point lies outside of any red points ($X^0_{i,j}$). This means we introduce extrapolation errors into our calculations which we want to minimize. To rectify this issue we can make a rule that if any state in $X^+_{i,j,k,l,m}$ has SiO_2 mass larger than a threshold, we simulate a slag tap on it. This is a very reasonable rule, since the furnace has

finite volume and the slag mass needs to be limited at any time instant inside it. We simulate a 6 ton slag tap where

$$X^+_{i,j,k,l,m} = X^+_{i,j,k,l,m} - \begin{bmatrix} 6000 \\ X^+_{i,j,k,l,m,1}/6000 \\ X^+_{i,j,k,l,m,2}/6000 \end{bmatrix}, \quad \text{if } X^+_{i,j,k,l,m,1} \geq \mathrm{SiO}_{2,\mathrm{m,max}} - 500 \tag{10-6}$$

Figure 10-3 shows that using the transformation from (10-6) we transform the yellow region into the green region. This means that now all the X^+ states lie in the same region as the X^0 states, so we will not require extensive extrapolation.

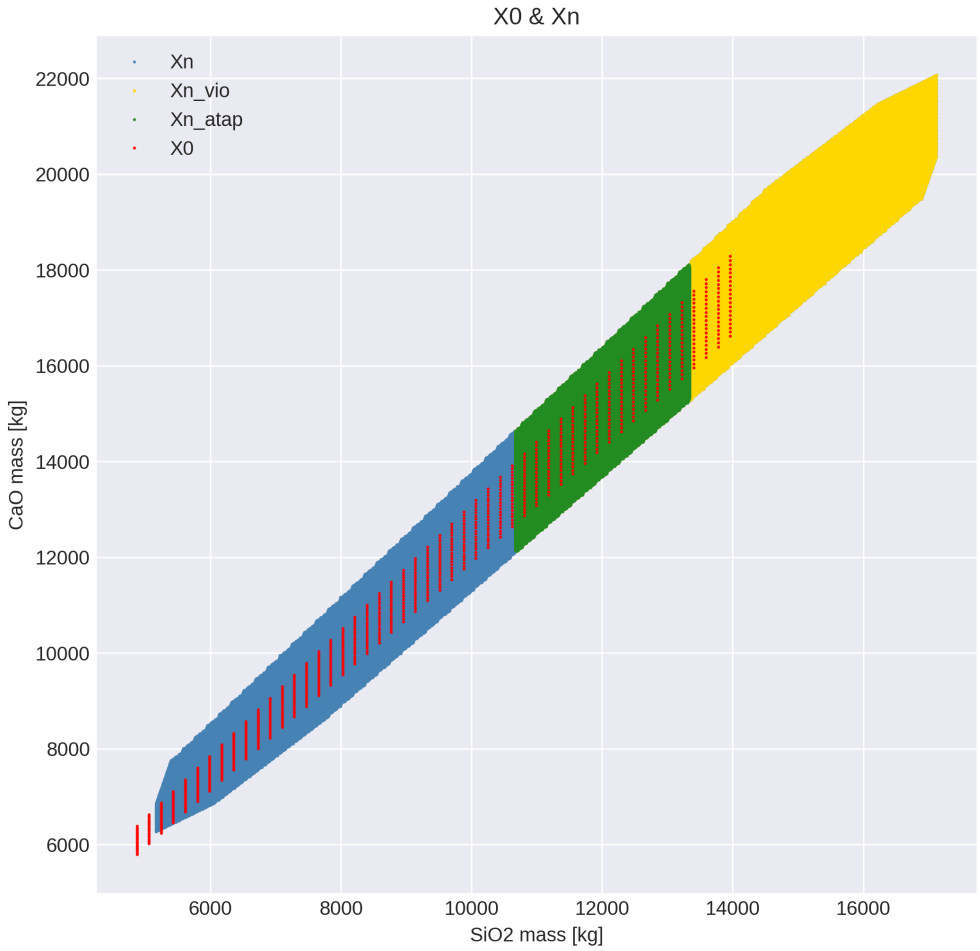

Figure 10-3: In blue X^+ states, in red X^0 states, in yellow the X^+ states that violate the SiO_2 mass condition, and in green the yellow states after we simulate a slag tap on them.

10-1-3 Running cost function

Our running cost function will tell us what is the cost of being in a certain state x and applying actions u_f and control action u_c. Since keeping the plant in the optimal operating condition is paramount, applying control input u_c will incur zero cost. Applying ore and coal inputs u_f will also incur zero cost since our task is hot metal production which is not possible without ore and coal inputs (u_f). This means our running cost function is only dependent on the current plant state x. More precisely we will only penalize deviations from our target basicity in the plant states.

$$g(x) = g\left(B_{2,\text{ref}} - \frac{x_2}{x_1}\right) = \begin{cases} 330 & x < 1.2 \\ 132000(1.25 - x_2/x_1)^2 & 1.2 \leq x \leq 1.25 \\ 400000(1.25 - x_2/x_1)^2 & 1.25 < x1.3 \\ 1000 & 1.3 < x \end{cases} \tag{10-7}$$

This cost function topology was devised based on the operator experience and recommendations of plant operation. If we are below 1.2 or above 1.3 basicity then we are outside of the plant operating region. These regions are both highly penalized, although being above 1.3 is more highly penalized since it is dangerous for the plant operation because slag foaming can occur in this region which can damage the plant. The most optimal point of our cost function is the 1.25 target basicity state while we penalize any deviation from it quadratically both if we are above or below it. The above region is penalized approximately 3 times more, since high basicity regions are dangerous for operation so we would really prefer to be below or at 1.25 basicity.

Figure 10-4 shows the chosen running cost function $g(x)$:

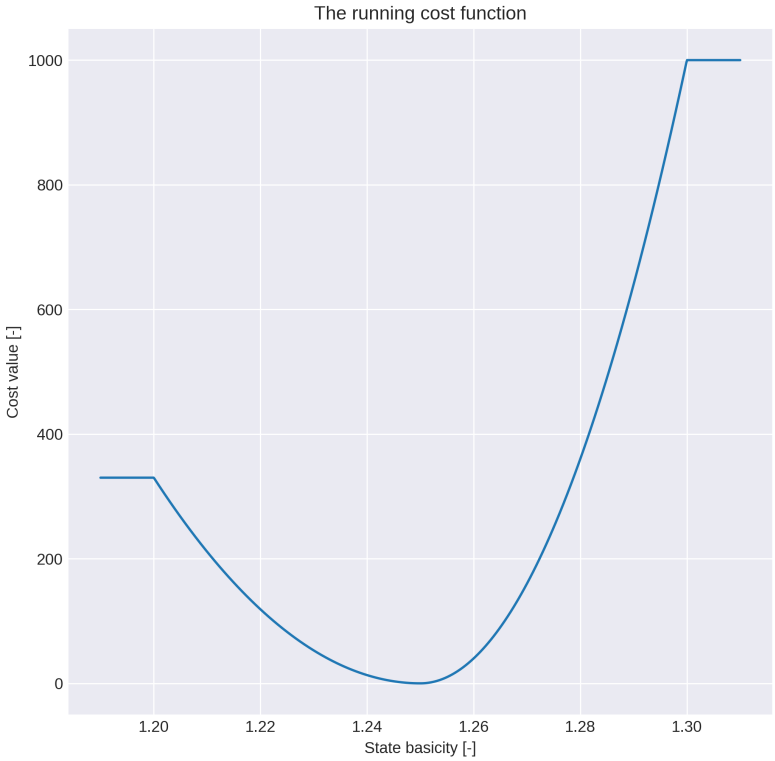

Figure 10-4: Running cost function $g(x)$.

10-1-4 Interpolating the value function

In this section we will look at how multilinear interpolation will work. We need this functionality since we only know the value of the value function at X^0 but we want to calculate $V(X^+)$.

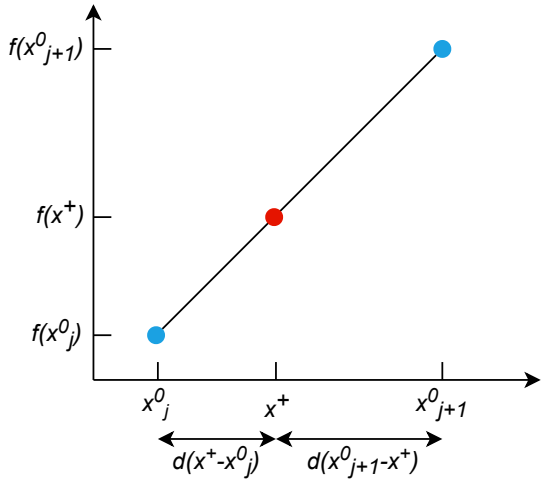

Figure 10-5: The value function is only known at discrete locations, but sometimes we need to know its value at a different location so we have to use interpolation

Figure 10-5 shows the one dimensional case of linear interpolation. We can calculate the value $f(x^+)$ of the red point x^+ from the known values $f(x^0_j)$ and $f(x^0_{j+1})$ as,

$$\begin{aligned} f(x^+) &= (1 - d(x^+ - x^0_j))f(x^0_j) + d(x^+ - x^0_j)f(x^0_{j+1}), \\ \text{where,} \quad & d(x^+ - x^0_j) + d(x^0_{j+1} - x^+) = 1. \end{aligned} \tag{10-8}$$

where $d()$ is a distance measuring function between points such that for $\forall j: \quad d(x^0_{j+1} - x^0_j) = 1$. For linear interpolation to work first we have to find the nearest grid points x^0_j, x^0_{j+1} to our location x^+. This is done by Algorithm A.14 where we use the fact that our grids have equidistant points. This speeds up the interpolation process significantly, because instead of searching for the two nearest grid points we can analytically find the nearest points just from our location and the known grid parameters. Although ideally we would like to interpolate all x^+ values sometimes this will not be possible because not all X^+ points lie inside the $\hat{\mathbb{X}}^0$ grid from Figure 10-3. This is the result of our model dynamics so it is something we cannot change. In case x^+ lies outside of the grid we can still use the same analytical method to calculate $V(x^+)$ but in this case we will be extrapolating the value from the nearest two know grid points. We calculate the interpolated value of $V(x^+, u_f)$ by Algorithm A.15 with a 2D interpolation for the SiO_2 and CaO states. If we interpolate on multidimensional space we use multiple interpolations sequentially along all state space dimensions as, (for 2D interpolation)

$$\begin{aligned} f(x^+, y^+) &= (1 - d(x^+ - x^0_j))f(x^0_j, y^+) + d(x^+ - x^0_j)f(x^0_{j+1}, y^+), \\ f(x^0_j, y^+) &= (1 - d(y^+ - y^0_k))f(x^0_j, y^0_k) + d(y^+ - y^0_k)f(x^0_j, y^0_{k+1}), \\ f(x^0_{j+1}, y^+) &= (1 - d(y^+ - y^0_l))f(x^0_{j+1}, y_l) + d(y^+ - y^0_{l+1})f(x^0_{j+1}, y^0_{l+1}), \end{aligned} \tag{10-9}$$

which is,

$$f(x^+, y^+) = (1 - d(x^+ - x_j^0)) \left[\left(1 - d(y^+ - y_k^0)\right) f(x_j^0, y_k^0) + d(y^+ - y_k^0) f(x_j^0, y_{k+1}^0) \right] + d(x^+ - x_j^0) \left[\left(1 - d(y^+ - y_l^0) f(x_{j+1}^0, y_l) + d(y^+ - y_l^0) f(x_{j+1}^0, y_{l+1}^0)\right)\right]. \tag{10-10}$$

The order of interpolating along the dimensions in our state space will be important in our case because some of our grids are dependent on other grids. This means first we interpolate along the free dimension's grid (x from (10-9)) and then on the dependent dimension's grid ($yfrom(10-9)$). We can calculate the neighbouring points along the dependent grid also analytically since the dependent grid is also equidistant.

10-1-5 Calculate the expectation

Our next task if to define how we calculate

$$\int V(\tilde{x}, u_f) p(\mathrm{d}\tilde{x} | x, u_f, u_c) = \mathop{\mathbb{E}}_{w} \left[V(x^+(x, u_f, u_c) + w, u_f) \right] \tag{10-11}$$

The integral here equals the expected value of the value function at the predicted state distribution $x^+ + w$ if we start from state x and apply inputs u_f and control action u_c. Here we have two choices on how to calculate the expectation:

- Monte Carlo simulation: sample noise distribution many times then average
- Integrate over a discretized model error distribution

We can calculate the expected value of the value function at the predicted state distribution using 1000 samples of the model error distribution w_i as,

$$\mathop{\mathbb{E}}_{w} \left[V(x^+ + w, u_f) \right] = \frac{1}{1000} \sum_{i=1}^{1000} V(x^+ + w_i) \tag{10-12}$$

This method will have a higher probability producing a good estimate of the true expected value the higher number of times we sample the noise distribution. We usually use the Monte Carlo method when our probability distribution is high dimensional and it is computationally not feasible to integrate over a sensibly dense discretized grid of model error distribution. Fortunately this is not the case with us, since our model error distribution is only two dimensional.

The second approach is using the noise distribution sample set we already exported with Algorithm A.10 and shown on Figure 8-4. By using this sample noise dataset $w_i \in \mathbb{W}$ we can estimate the expectation as follows:

$$\mathop{\mathbb{E}}_{w} \left[V(x^+ + w, u_f) \right] = \sum_{i=1}^{n} p(w_i) V(x^+ + w_i, u_f) \tag{10-13}$$

where $p(w_i)$ is the probability of w_i occurring which we stored in the heights array, and w_i is a vector noise sample like [0; SiO2 model error, CaO model error].

10-1-6 Value iteration

With the above section we have fulfilled every requirement for the value iteration process to start. We can substitute (10-13) into (10-1) and arrive at our value iteration equation:

$$\Gamma(V_{n+1})(x, u_f) := \max_{u_c \in \hat{\mathbb{U}}_c} g(x) + \beta \sum_{w_i \in \hat{\mathbb{W}}} p(w_i) V_n(x^+(x, u_f, u_c) + w_i, u_f). \tag{10-14}$$

The value iteration is calculated in the code by Algorithm A.16. It is usually enough to run the algorithm for 25-30 iteration until convergence is achieved which is defined as,

$$|V_{n+1}(x, u_f) - V_n(x, u_f)| \leq \epsilon \tag{10-15}$$

Then we can calculate the controller as,

$$u_c(x, u_f) := \operatorname*{argmax}_{u_c \in \hat{\mathbb{U}}_c} g(x) + \beta \sum_{w_i \in \hat{\mathbb{W}}} V_{n+1}(x^+(x, u_f, u_c) + w_i, u_f). \tag{10-16}$$

The practical aspects of the value iteration calculation are the following: We used a highly parallelized framework calculating the value function on 16 threads, while also using numba which pre-compiles python functions into machine code that can be run by the python interpreter and can usually replace for loops used in python. With these optimization it is possible to reduce the calculation time of the static value function by orders of magnitude.

10-2 Controller

Now that we have obtained the controller table from (10-16) what we have calculated is essentially the optimal control input u_c^* for all grid points from our starting states $\hat{\mathbb{X}}^0$ and fixed inputs $\hat{\mathbb{U}}_f$. In order to actually obtain a controller from the controller table we have to use multilinear interpolation to calculate the optimal control input to any state-fixed input combination that do not lie on the grid $\hat{\mathbb{X}}^0$ and $\hat{\mathbb{U}}_f$. We will be using the same multilinear interpolation technique as in section 10-1-4 but now we also interpolate along the u_f axes. Thus the controller implementation is shown by Algorithm A.21.

An important property of the obtained controller is that by using a model structure which is time independent (mass based, time is not important) we make sure that our controller will also be time independent. This means our controller will output how much lime has to be injected until the next slag tap (state measurement), based on the current plant state and the ore and coal input amounts that will be injected until the next tap, but it is unimportant how much time elapses until the next slag tap. Now a possible problem with this framework is that we use a static error distribution to model our system error during controller synthesis. If we want our controller to work for a wide variety of possible u_f (ore and coal injection rates) then our static error distribution no longer encompasses the true model error well. This is because for example the possible error on the model prediction should be much smaller when using an ore input of 100 kg than if we use an ore input of 20 000 kg since most model inaccuracies come from ore mixture variation or B matrix inaccuracy. This means our controller synthesis approach will not yield an optimal solution.

To fix this issue a possible solution could be deriving a model error distribution that is dependent on the total amount of injected material. This would be a rational choice since if one parameter from the B matrix of our model is for example off 1% then the resulting model error will be 1% times the injected ore, coal or lime amount depending on which element of the B matrix has the deviation.

Our next task will be to validate if our controller performs up to expectation.

10-2-1 Controller validation

The controller validation is one of the hardest tasks in the project. The best way to validate the controller would be to just connect it up to the plant and see if the performance is up to standard. Unfortunately this is not possible as a first test since the plant have a very narrow operating region and in case our controller does not perform optimally we can end up outside the operating region possibly causing damage to a multi million dollar plant.

In order to sidestep this issue we can use simulations of what would happen in the plant in case we used the controller to control it. We can do this by using our system model, but we also have to take into account that our model is not perfect and can have model errors. Our task will be to validate a controller on a statistical basis. For example we can easily calculate

what would our system model predict as next states if we used the controller as,

$$\hat{x}(k+1) = f(x(k), u_f(k), u_c(x(k), u_f(k))) \tag{10-17}$$

where

- $x(k+1) \in \mathbb{X} \subset \mathbb{R}^3$ is the predicted next state vector of the system model at time $k+1$ (model prediction),
- $x(k) \in \mathbb{X} \subset \mathbb{R}^3$ is the current state vector of the plant at time k (measurements),
- $u_f(k) \in \mathbb{U}_f \subset \mathbb{R}^2$ is our fixed plant inputs vector (measurement),
- $u_c(k) \in \mathbb{X} \times \mathbb{U}_f \to \mathbb{U}_c$ is our controller advice at time k (controller output),
- $f \in \mathbb{X} \times \mathbb{U}_f \times \mathbb{U}_c \to \mathbb{X}$ is our system model.

The prediction $x(k+1)$ is the state the real plant would transition into in case we used the controller with it and the system model were perfect. To incorporate the effect of the model uncertainty we can add sampled random noise to our predicted next plant state. Since we already know the model prediction error distribution we can use this information and sample our random noise from this distribution. This way we make sure we are adding a realistic noise to our predicted next states.

$$\tilde{x}(k+1) = f(x(k), u_f(k), u_c(k)) + \omega, \tag{10-18}$$

where $\tilde{x}(k+1)$ is the noisy prediction of the next plant state and ω is a random sample from our model error distribution from Algorithm A.10 (Figure 8-4). If we create the noisy predictions for all time steps ($k = 0, 1, \ldots$) we can compare the performance of the operator and controller. If we use this simulation framework we are only using the model as a one ahead predictor. This means we are comparing how well does the controller and the operator react to deviations in the slag basicity in one time step.

First lets compare the actions of the operator and the controller taken during the one ahead simulation. Figure 10-6 shows this comparison for a certain plant run dataset. Please note we used here a min max scaled dataset in order to protect the intellectual property of TataSteel.

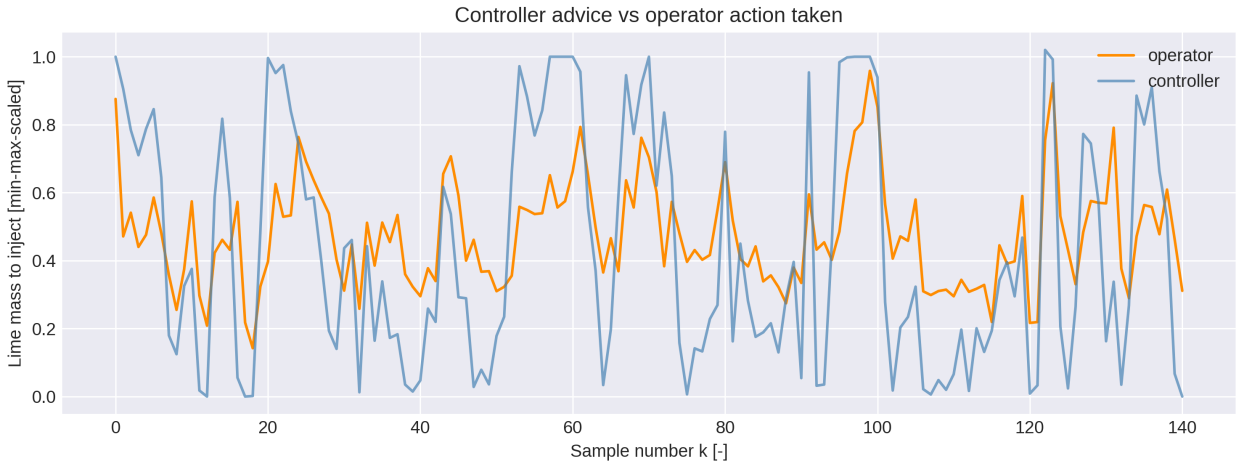

Figure 10-6: Control action taken by operator versus advised by controller in each time step k

We can see from Figure 10-6 that the controller advice has a similar trend to the operator control actions which is expected since both are reacting to the same plant state changes and fixed input fluctuations. We can also see that the controller is more aggressive and often applies a control action which is more toward the extremities (low and high) which is also as expected since the operators often do not try to correct basicity deviations in one time step but rather try slow corrections toward the optimal value instead in multiple time steps. We can also compare the graphs of the plant state evolution $x(k)$ and the model predictions with the control action $\tilde{x}(k)$ on Figure 10-7. We also used a min max scaled dataset here in order to protect the intellectual property of TataSteel.

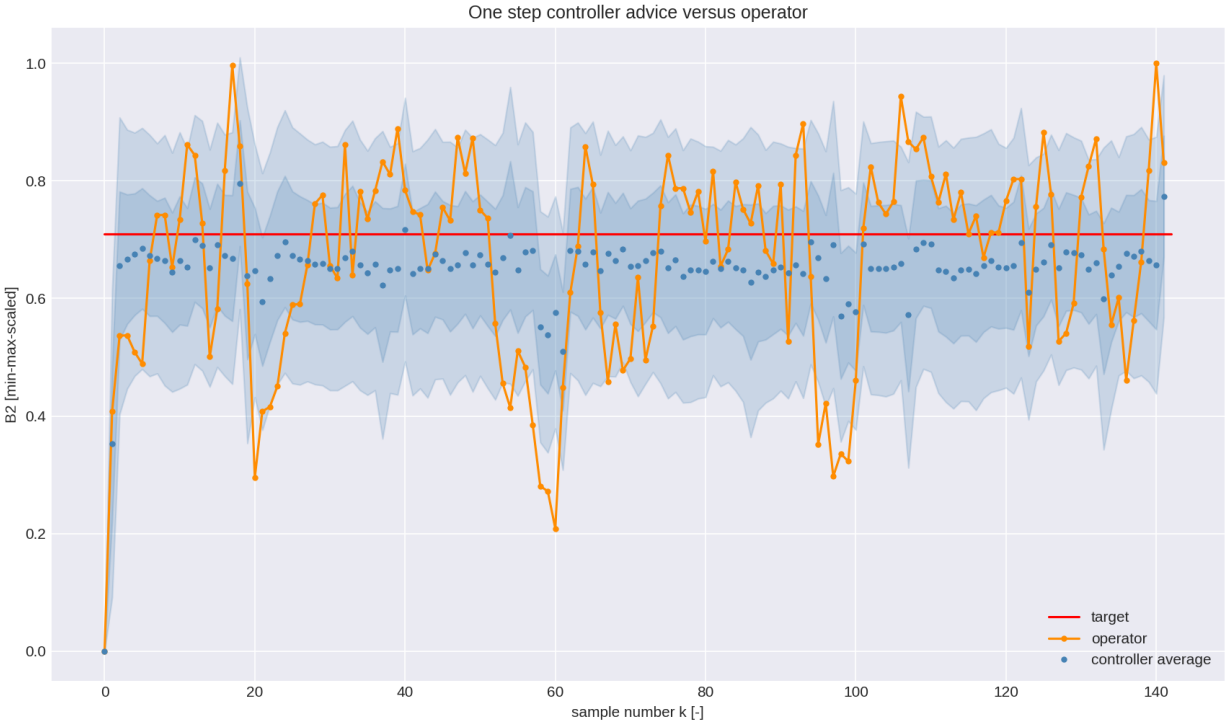

Figure 10-7: Comparison of plant state evolution with operator control versus where the controller would steer the plant.

We can see on Figure 10-7 the measured plant basicity evolution $x(k)$ with operator control in orange and we can also see the blue data points where the controller would steer the plant $\tilde{x}(k+1)$ from the previous measured state $(x(k))$. We used here only dots since the dots do not represent a state evolution of the plant, merely indicate where the plant would end up at if we applied the controller recommendation from the previous measured state and the system model was prefect. The filled blue areas indicate how certain is our model that we end up at a blue dot. The darker region is one standard deviation above and below our model prediction, and the light area cover a plus minus two standard deviation range of our model error. This means if we apply control action $u_c(x(k), u_f(k))$ starting from $x(k)$ then we have a probability of $\approx 68.27\%$ of that our plant ends up in the dark blue area and $\approx 95.45\%$ probability to end up inside the light or dark blue areas and on average we will end up at the blue dot $\overline{\tilde{x}}(k+1)$, which is:

$$\overline{\tilde{x}}(k+1) = \frac{1}{n}\sum_{i=1}^{n} f(x(k), u_f(k), u_c(k)) + \omega_i \tag{10-19}$$

Now we can see on our graph that the controller on average will probably be better at controlling the plant than the operator since the blue dots lie near the 1.25 target basicity line but we can also see that we have a large model error distribution on the plant basicity

which makes it hard to decide if in fact the controller really performs better than the operator. To have a better tool for judging the controller performance we can do the following: Calculate 1000 times $\tilde{x}(k+1)$ from (10-18) for all time steps k which we will denote as $\tilde{\boldsymbol{x}}_j$ for the j-th calculation (from the 1000). This way we can get a statistical baseline on what the result of our controller advice would look like with imperfect model error. Next we can use the cost function from (10-7) to evaluate the operator performance $\boldsymbol{x} = \begin{bmatrix} x(1) & x(2) & \dots & x(n) \end{bmatrix}$ as $g(\boldsymbol{x})$ and similarly we can evaluate the performance of the plant controller advices as $g(\boldsymbol{x}_j)$. Next we can look at the best, average and worst controller advices from the 1000 simulations.

operator score	$g(\boldsymbol{x})$	19706
best controller advice simulation	$\min_j g(\tilde{\boldsymbol{x}}_j)$	7974
average controller advice simulation	$\text{mean}_j\, g(\tilde{\boldsymbol{x}}_j)$	11479
worst controller advice simulation	$\max_j g(\tilde{\boldsymbol{x}}_j)$	15962

Table 10-2: Controller advice simulations scores versus operator score

In Table 10-2 we can see that on average our controller performs better than the operator in controlling the plant from given plant states toward the basicity target in one step. We can also see even for the worst run from 1000 simulation the controller still outperforms the operator in controlling the plant toward the target basicity in one time step. This is as expected since the operators mostly try to use slower corrective control actions during operation and steer the plant toward the target basicity using multiple time steps because they do not know the plant state response to the corrective control actions, thus they stay on the side of caution and stick to more average lime injection amounts instead of aggressive corrections. Now we know the controller performs better with a high probability than the operator in steering the plant toward the target basicity in one time step. The next thing to check is if the controller can maintain stable operation of the plant by using multiple sequential control actions. This means we have to simulate the plant behaviour for multiple time steps forward not just one step ahead as previously for Figure 10-7 and Table 10-2. This can be done similarly to (10-18) but instead of using measurements $x(k)$ we will need to use the previous simulated states $\tilde{x}(k)$:

$$\begin{aligned} &\tilde{x}(0) = x(0), \\ &\tilde{x}(k+1) = f(\tilde{x}(k), u_f(k), u_c(\tilde{x}(k), u_f(k))) + \omega_i. \end{aligned} \tag{10-20}$$

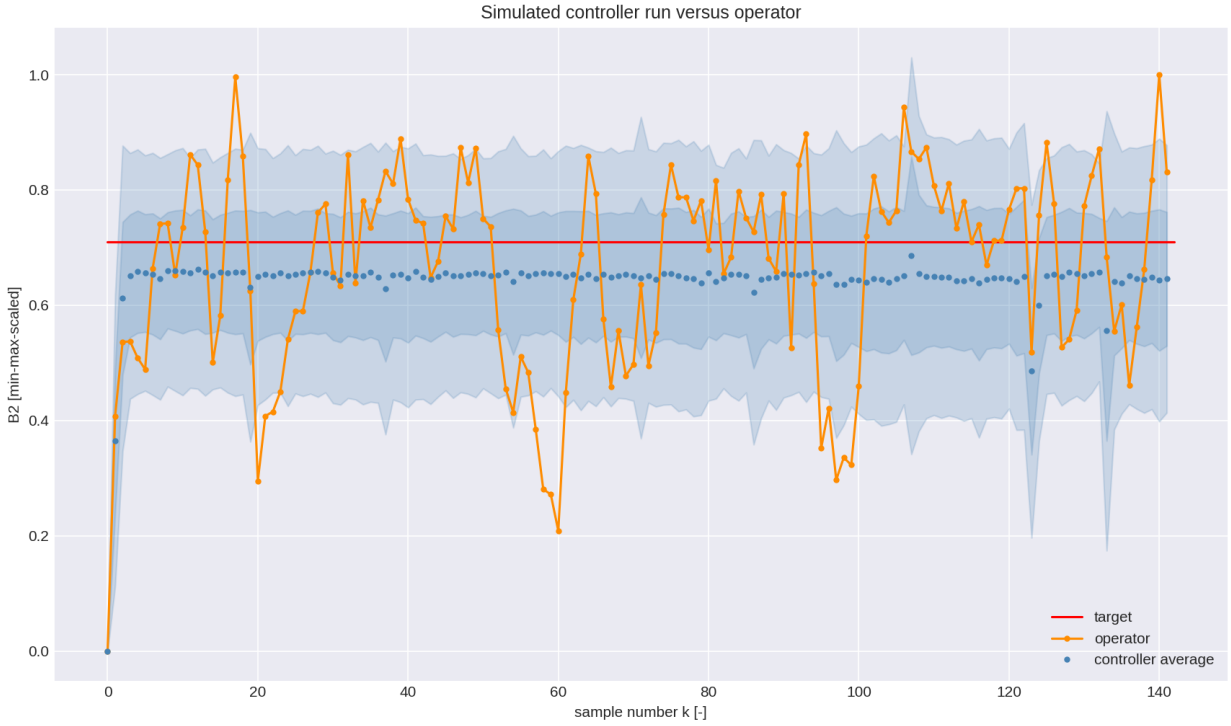

Figure 10-8: Simulated controller run versus operator run (same fixed inputs used $\boldsymbol{u}_{12}$).

Figure 10-8 shows the operator run measurements in orange, the average controller state in blue and the one standard deviation of the controller state with respect to the model error distribution as a dark blue region, and the two standard deviation region as a light blue region. We can see our controller on average will converge to the basicity target of 1.25 well but due to the model errors will deviate from the basicity target significantly per simulations because of the model error uncertainties as shown by Figure 10-9. We can also see that the controller average is consistently slightly bellow the target basicity. This id due to the shape of our cost function from Figure 10-4 and the size of our error distribution from Figure 8-4. The controller will try to keep the basicity level slightly below the target value in order to have a low expected cost for any arising model errors, because we penalize model error that overshoot the 1.25 target basicity line much more than model error below the target. The two Figures: 10-8 and 10-9 were also min max scaled to protect the intellectual property of TataSteel.

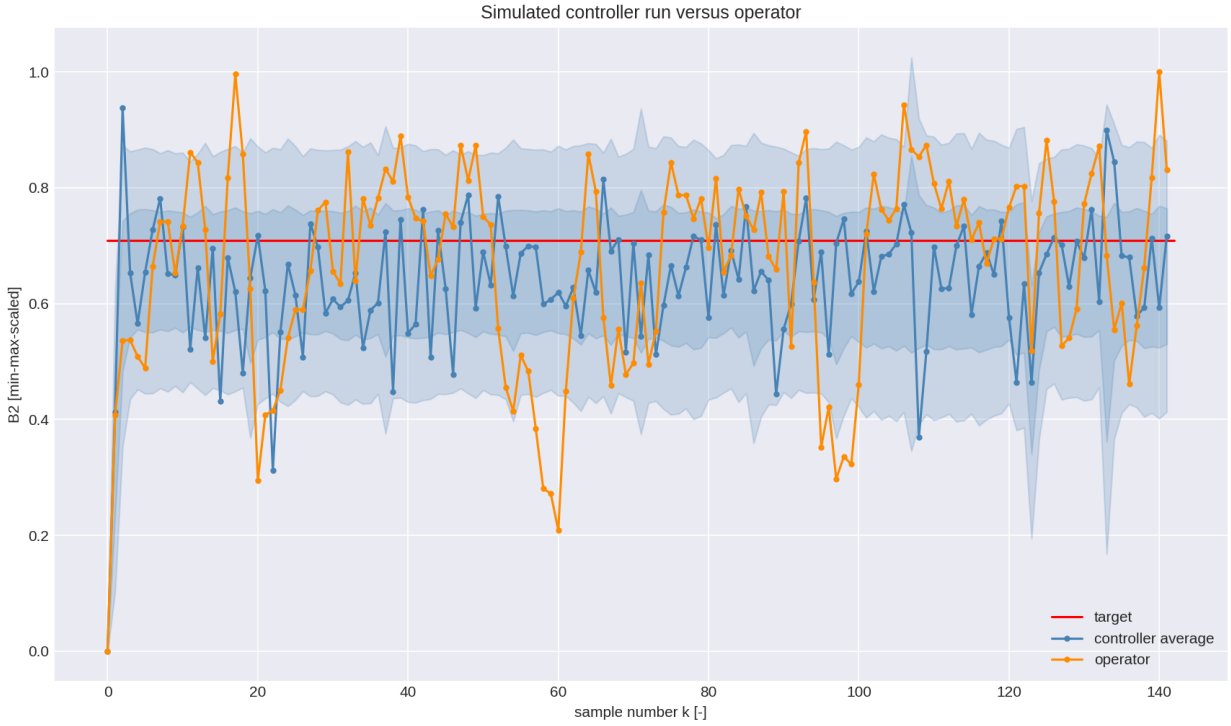

Figure 10-9: One controller simulation compared to the operator run measurements.

As we can see on Figure 10-9 the controller although the model has a large model error distribution still manages to keep the basicity of the plant near the target better than the operator. Lets compare again the scores of 1000 simulations with the operator run score:

operator score	$g(\boldsymbol{x})$	19706
best controller advice simulation	$\min_j g(\tilde{\boldsymbol{x}}_j)$	7211
average controller advice simulation	$\text{mean}_j\, g(\tilde{\boldsymbol{x}}_j)$	11395
worst controller advice simulation	$\max_j g(\tilde{\boldsymbol{x}}_j)$	18399
simulation from Figure 10-9 score	$g(\tilde{\boldsymbol{x}}_{421})$	8130

Table 10-3: Controller advice simulations scores versus operator score

In Table 10-3 we can see the controller on average still outperforms the operator significantly when we simulated the plant state not just one step ahead but for all step of a plant run fixed input dataset $\boldsymbol{u}_f$. We can also see in the worst case the controller has comparable performance to the operator from 1000 simulations. It would be even possible to significantly improve these controller results by slightly reducing the model error distribution if we are able to build a better system model in the future. This is certainly possible since now we are using a very crude system model and Tata Steel has a more precise model for plant operation control which they are in the process of simplifying in order to be used with simulation.

There is one more plot we can examine as it might contain valuable information of our controller. We can check the controller advice for certain 2D slices of our controller table as shown by Figure 10-10 .

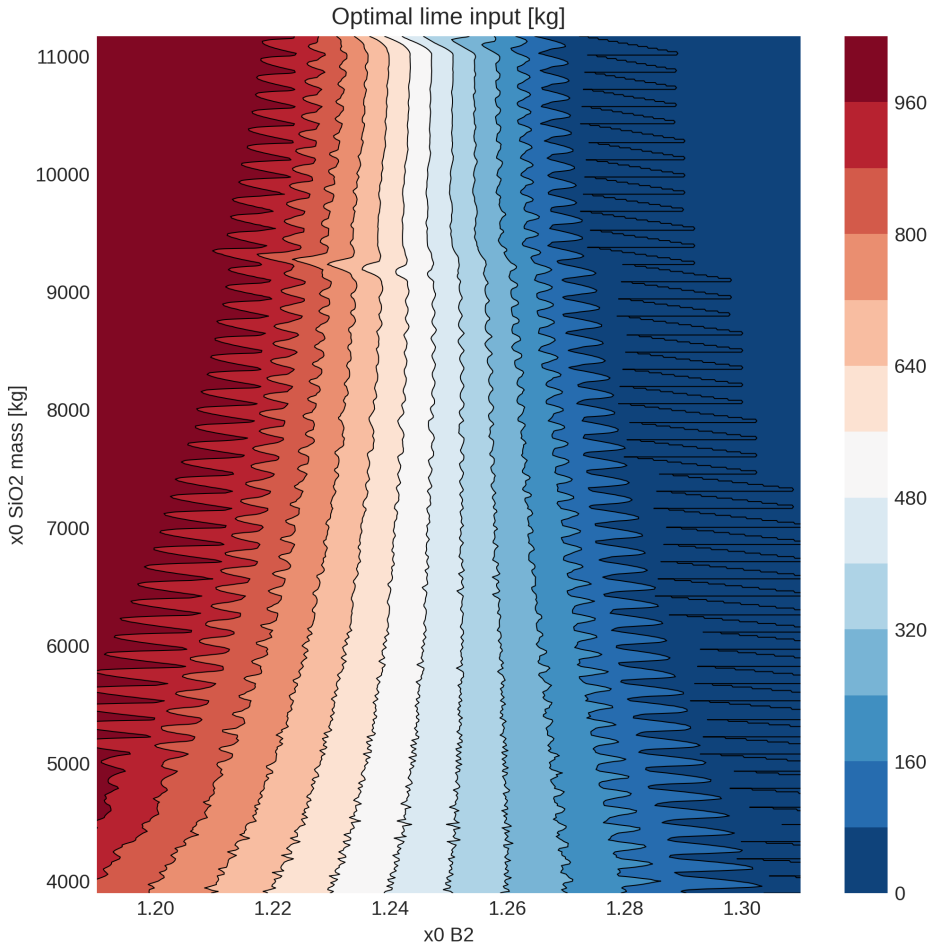

Figure 10-10: Controller advice plot for static fixed inputs u_f = constant.

We can see on Figure 10-10 that if we fix our fixed inputs to a certain constant $u_f = \begin{bmatrix} 26000 & 14000 \end{bmatrix}^\top$ then the amount of lime we need to inject is actually linearly dependent on the current slag basicity (plant state). This can be seen because the colored stripes of the contour plot have a similar thickness. We can also see that as the SiO_2 mass in the furnace increases we need to apply use a more aggressive control. This can be seen from how the colored stripes narrow down towards the top. This means if we are at a high SiO_2 mass we will need to apply large control actions for smaller deviations of the basicity (B_2) compared

to low SiO_2 mass states. This means as a future work we can try to approximate the value function for a fixed u_f with a smooth function instead of multilinear interpolation. We can also see periodic waveform of Figure 10-10. These are likely caused by the states grid and multilinear interpolation that we decided to use. Figure 10-11 shows the same plot as Figure 10-11 but with different axes.

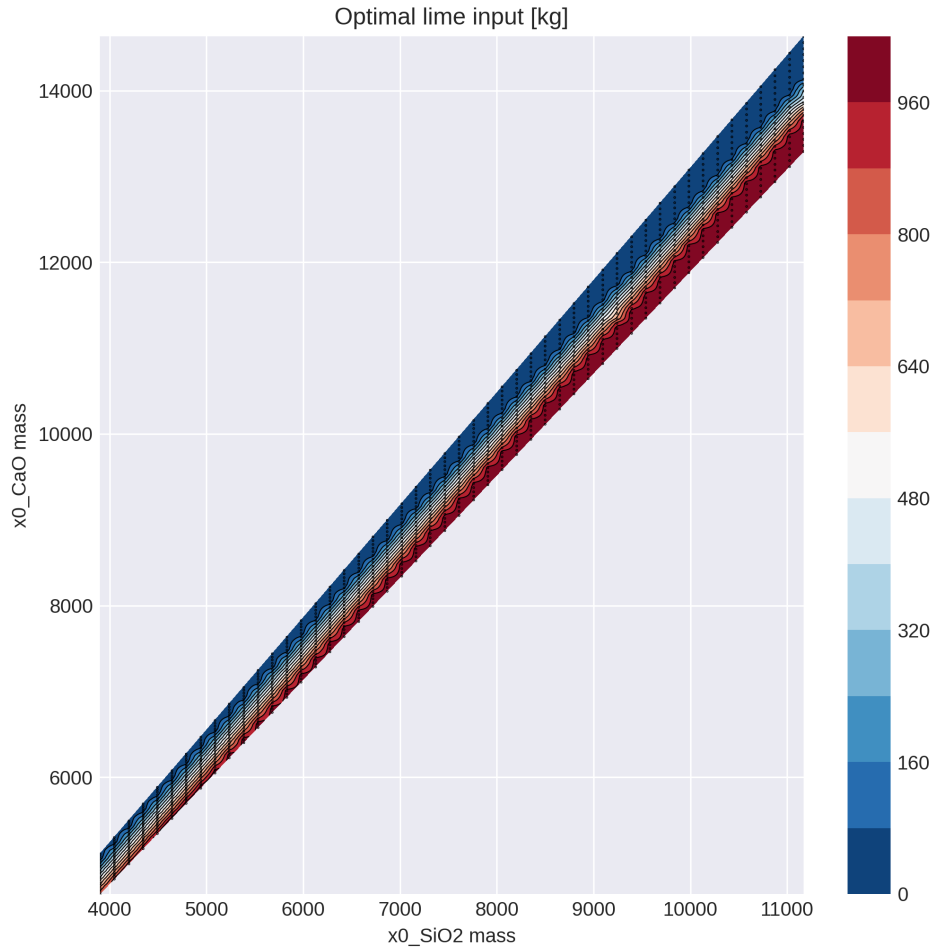

Figure 10-11: Controller advice plot for $u_f = \begin{bmatrix}26000 & 14000\end{bmatrix}^\top$.

The states grid that we used $\hat{\mathbb{X}}^0$ is indicated on Figure 10-11 as black dots. We can see between each vertical line of black dots there is one wave visible. A different parameterization of the state space would be advisable in future work as to minimize this undesirable wave effect which is simply an artifact of multilinear interpolation and our grid shape. We used this state space grid $\hat{\mathbb{X}}^0$ because it enabled analytic calculation of the neighboring points in the multilinear interpolation.

If this controller is put into production, as future work a model error checking script would be advised to run beside it. The controller will only give valid advice until the model errors lie within the previously calculated model error distribution. When this is not the case the controller should be recalculated with the updated model error distribution. In this case refitting the system model would be also a good idea.

Chapter 11

Conclusions

This thesis discussed the basicity control of HIsarna: an experimental smelting furnace. The novelty of the work carried out is solution to the stochastic infinite horizon dynamic programming control problem that was solved to control the basicity of the furnace. Hisarna is a pilot plant, which is the first of its kind in the world and it requires more complex control methods than traditional blast furnace technology since there is a more complicated process inside the furnace that needs constant attention for successful operation. Automating the basicity control of this plant is one step toward production operation of this pilot plant, which would mean 30% carbon emission reduction compared to traditional blast furnaces and large cost savings because HIsarna does not require a pellet plant preprocessing of the input materials.

The thesis discussed the following topics: First we created a grey-box nonlinear system model which enabled us to model the relevant plant behaviour for different ore mixtures. We fitted the model parameters by a parameter search method that was guaranteed to find a solution near the global optimum if certain smoothness conditions of the cost function were met. Next we evaluated our system model extensively by examining the error histograms on the validation dataset and the correlation between the model errors and the plant inputs and states. We concluded that our system model is adequate for controller synthesis since we observed zero mean model errors and no or low correlation coefficients. For controller synthesis we used Value Iteration. We needed a continuous value function because our system model had a continuous state space. We selected multilinear approximation of the continuous value function as our approach since we did not have any knowledge of the expected shape of the value function. We used the same multilinear interpolation technique to derive a continuous controller.

In the controller evaluation part of the thesis the challenge was how to evaluate the expected performance of the controller on the real system without actually operating the furnace with the controller. It is necessary the validate the controller virtually first because HIsarna is a multi-million dollar project and incorrect operation of the plant can damage the furnace. We settled on comparing the performance of the recorded operator runs versus the simulated

controller performance with our stochastic system model in two performance metrics: First we compared how the operator and the controller adjust to deviations of the plant basicity in one time step which yielded favorable results for the controller. Then we compared the operator and the controller performance starting from the same initial state but simulating for the same number of time steps ahead as we have recorded data. Again on average the controller outperforms the operator according to our running cost function that we we used in the dynamics programming formulation and in the worst case from 1000 simulations has comparable performance to the operator. From these results we conclude that the achieved controller has good potential for real world application but we point to several future work improvement points that can greatly increase the controller performance such as using a more accurate system model or using a different grid for the value function parametrization or smooth approximation of the value function.

Appendix A

Code snippets

This appendix contains code snippets used during the thesis.

Algorithm A.1: Data read and label convert

```
# Slag inv. data & Run Data
data = pd.read_csv(workdir+"/data/Historian_Data_Consumption.csv", sep=";
    ",decimal=",")
S_data = pd.read_csv(workdir+"/data/Slaktap_Data_run3_2.csv", sep=",",
    decimal=".")

# column name translation file
data_H2S = pd.read_csv(workdir + "/data/columns_H2G.csv", sep=";")

# Create translation dictionaries to rename columns in dataset to '
    Standard'
dict_H2S = dict(zip(data_H2S["His_database"], data_H2S["G_names"]))
data.rename(columns=dict_H2S, inplace=True)
```

Algorithm A.2: Shift timelables of Slagtap data

```
# Reformat date columns as needed, round to minutes,
data["DateTime"] = pd.to_datetime(
    data.DateTime.str.replace("T", " ", regex=False).str.replace(
        "Z", ".000", regex=False) ).round("min")

# And Slag data is different timezone (substract 2h)
S_data["DateTime"] = ( pd.to_datetime(S_data.DateTime).round("min")
                       - np.timedelta64(2, "h") )
```

Algorithm A.3: Reformat data values

```
data["slag_inventory_calc"] = data.slag_inventory_calc * 1000
data["Slag_inv"]    = data.Slag_inv * 1000
data["CaO_rate"]    = data.CaO_15min * 4
data["Coal_rate"]   = data.Coal_rate_1 + data.Coal_rate_2
```

```
data["M_Slag_SiO2"] = data.M_Slag_SiO2 / 100
data["M_Slag_CaO"]  = data.M_Slag_CaO / 100
```

Algorithm A.4: Transform slag inv. dataset

```
# get corrected amount of tapping
slag_mptap_opt[0] = np.concatenate(( [ S_data.Slag_mptap.values[0] ],
                                     ( S_data.Slag_inv_corr.values[:-1]
                                       + S_data.HCM_make.values[1:]*1000
                                       - S_data.Slag_inv_corr.values[1:] )
                                         ))

# get intersection of datasets
s_ind, slag_m = np.intersect1d( S_data.DateTime.values, data.DateTime.values,
                                assume_unique=True, return_indices=True )[1:3]
s_ind      = s_ind[slag_m < end]   # intersect indexes of S_data (bool)
slag_m     = slag_m[slag_m < end]  # tapping times in 'data indices'

# chop the mptap-s to length
slag_mptap_opt[0] = slag_mptap_opt[0][s_ind]

# Get slag inventory meas. uncorrected, and corrected
Slag_inv_m_opt[0] = S_data.Slag_inv_corr.values[s_ind] + slag_mptap_opt[0]
```

Algorithm A.5: Create SiO2, CaO and input amounts measurements

```
def meas_at_t(dat, slag_m): # get measurement at only slag measure times
    return dat[slag_m]

def create_U(data, slag_m):
    u = np.zeros((len(slag_m)-1,3))
    for i in range(len(slag_m)-1):
        t0 = slag_m[i]
        t1 = slag_m[i+1]

        sum_o = np.sum(data.Ore_rate.values[t0:t1]) # injected ore between t0 and t1
        sum_c = np.sum(data.Coal_rate.values[t0:t1])
        sum_l = np.sum(data.CaO_rate.values[t0:t1])

        u[i,:]=[sum_o, sum_c, sum_l] # create vector

    # There are 60 measurements in an hour and each is [kg/h], we return [kg]
    return u / 60

# Get measurements of CaO, SiO2 at taps (~approx.)
m_offset = 70
CaO_f    = meas_at_t(CaO_new,  slag_m + m_offset)
SiO2_f   = meas_at_t(SiO2_new, slag_m + m_offset)

```

```
# Calc. [ore[slag_m[i]:slag_m[i+1]] for i in range(len(slag_m))]
U_f      = create_U(data, slag_m)
```

Algorithm A.6: Grid search algorithm

```
def run_grid_searchprop(sys, Th0, Y, sim, F, S_range, prec,
    prog_bar_build,
                        scaler=[1,1,1], allow=0, u_dim=0):
    """Find the best Theta with a grid search"""
    # S_range = [[xa,xb],[ya,yb],[za,zb]]
    # Create grid points
    Th_grid = np.array(np.meshgrid( np.arange(S_range[0,0], S_range[0,1]
        + prec, prec),
                                    np.arange(S_range[1,0], S_range[1,1]
                                        + prec, prec),
                                    np.arange(S_range[2,0], S_range[2,1]
                                        + prec, prec)
                                   )).T.reshape(-1,3)
    Th_len = len(Th0)
    n = len(Th_grid)

    # Theta value for each grid point
    Th_arr   = np.outer( np.ones(n+1), Th0 ).astype('float'); # n+1 x
        Theta,
    # predicted X array
    X_arr    = np.zeros( (n+1, u_dim, len(scaler)) )          # n+1 x sim
         output
    # Cost value array
    cost_arr = np.zeros(  n+1 )                               # n+1 x
        cost

    # -------------SIMULATE n times-----------
    for i in range(1, n+1):
        Th_arr[i, np.where(allow)] = Th_grid[i-1]
        # save cost and sim output
        cost_arr[i], X_arr[i] = err(sys, Th_arr[i], Y, scaler, sim, F)

    return Th_arr, cost_arr, X_arr;
```

Algorithm A.7: Distribution fitting

```
mean, var  = stats.distributions.norm.fit(temp) # Fit normal distribution
    to error histograms
border = max(abs(temp))                         # Largest error
x = np.linspace(-border,border,200)             # linspace to plot normal
    distr.
fitted_pdf = stats.distributions.norm.pdf(x, mean, var) # calc. pdf
    values of normal dist.
```

Algorithm A.8: Discretizing the model error set.

```
lim = 5*np.max(abs(cov**0.5)) # limits for the integral
# grid of x,y points
X, Y = np.meshgrid(np.linspace(-lim,lim,1000),np.linspace(-lim,lim,1000))
```

```
# Create list from X,Y
xy = np.concatenate((X[:,:,None],Y[:,:,None]),axis=2).reshape((-1,2))
# Calculate the pdf at these xy grid points
Z = gauss_2D_pdf(xy,mean,cov).reshape((1000,1000))

zsum = Z.sum() # calculate the sum of our points
z = Z/zsum     # Correction so integral is equal to 1 (as len(x)->inf Z.sum->1)
```

Algorithm A.9: Calculate the value of the pdf at the 99-th percent contour line of the error distribution

```
t = np.linspace(0, z.max(), 1000) # contour heights array
integral = ((z >= t[:, None, None]) * z).sum(axis=(1,2)) # integrates the pdf
f = interpolate.interp1d(integral, t) # interpolates the integral
t_contours = f(np.array([0.99]))      # Select percentile line here, selects a contour line (height value)
```

Algorithm A.10: DP discrete error set

```
# Create grid with error values to be used in DP
zr = z.reshape((-1,1))
ret = np.array([xy[i] for i in range(len(xy)) if zr[i] > t_contours[0]])
lim2 = np.max(abs(ret), axis=0)  # Find the limits of the 99-th percentile line

X1, Y1 = np.meshgrid(np.linspace(-lim2[0], lim2[0], 20), np.linspace(-lim2[1], lim2[1], 20))
x1 = np.concatenate((X1[:,:,None],Y1[:,:,None]),axis=2).reshape((-1,2))

# Return the points inside the percentile line
ret = np.array([ [xi[0], xi[1], gauss_2D_pdf(xi, mean, cov)] for xi
                    in x1 if gauss_2D_pdf(xi, mean, cov)/zsum >
                        t_contours[0]])
ret[:,2] = ret[:,2] / ret[:,2].sum() # make them sum up to 1

bins = np.zeros((len(ret),4), dtype=x_type) # bins is 4D as X0
bins[:,:2] = ret[:,0:2].astype(x_type)
heights = ret[:,2].astype(x_type)
```

Algorithm A.11: Calculating predicted next states X^+ form X_0 and U

```
def Xn_calc(B, SiO2_bins, CaO_linsp, Ore_bins, Coal_bins, U_c_bins, SiO2_mean, x_type=np.float32):
    """Calc B2"""
    # ----------BU Calc.--------
    U_h = np.moveaxis(np.array( np.meshgrid(Ore_bins, Coal_bins, U_c_bins, copy=False)),
                      [0, 2, 3], [3, 0, 2])
    U_h[:,:,:,1] += U_h[:,:,:,0] / 1.83 # 1.83 = U_f[:,0]/U_f[:,1] -> Uf1 = Uf0 / 1.83
    BU = np.dot(U_h, B.T) # --------Checked

```

```
    # ----------X0 Calc.--------
    X_h = np.vstack( (np.ones(len(CaO_linsp), dtype=x_type) / SiO2_mean,
                      np.ones(len(CaO_linsp), dtype=x_type),
                      CaO_linsp)
                    ).T
    X0 = np.multiply.outer(SiO2_bins.T, X_h) # --------Checked

    # -----------Xn Calc.--------
    Xn = np.empty(( np.hstack((X0.shape[:-1],
                               BU.shape[:-1],
                               [4] ))
                  ), dtype=np.float32) # Create empty Xn array
    Xn[:,:,:,:,2] = np.array(range(Xn.shape[2]),dtype=np.int32)[None,
        None,:,None,None]
    Xn[:,:,:,:,3] = np.array(range(Xn.shape[3]),dtype=np.int32)[None,
        None,None,:,None]
    for i in range(X0.shape[0]):
        temp = X0[i,:,None,None,None,:] + BU[None,:,:,:,:]
        Xn[i,:,:,:,:,:2] = (temp[:,:,:,:,1:]).astype(np.float32)

    # Simulate slag tap if we go above a certain treshold of slag
        inventory
    for i in range(2):
        # Tap where SiO2_m > max(SiO2_measured) - 500kg
        idxes = np.where( Xn[:,:,:,:,0] > np.max(SiO2_bins) - 500 )
        # calc. multiplier for tapped indexes
        slg = Xn[idxes][:,0] / SiO2_mean
        slg = (slg-6000)/slg
        # modify tapped indexes with multiplier
        Xn[:,:,:,:,:,:2][idxes] = slg[:,None] * Xn[idxes][:,:2]
    print('States that violate border:', np.sum(Xn[:,:,:,:,0] > np.max(
        SiO2_bins)))
    return Xn, X0
```

Algorithm A.12: Running cost function: f_cost_R()

```
@vectorize(['float32(float32)'], nopython = True)
def f_cost_R(b2):
    """Running Cost function"""
    if b2>1.30 or b2<1.20:
        return 1000
    else:
        return 100000*(1.25-b2)**2
```

Algorithm A.13: Modified running cost function

```
@vectorize(['float32(float32)'], nopython = True)
def f_cost_R(b2):
    """Running Cost function"""
    if b2>1.30 :
        return 1000
    if b2<1.2
        return 250
    if b2<1.25
```

```
9           return   25000*(1.25-b2)**2
10      else:
11          return  100000*(1.25-b2)**2
```

Algorithm A.14: Function to find the index of neighbouring points

```
1  def id_from_m(x0, x0_min, x0_size, x0_len):
2       x0id = (x0 - x0_min) / x0_size
3       x0idx = max( min( int(np.floor(x0id)), x0_len - 2), 0)
4       x0idp = x0id - x0idx
5       return x0idx, x0idp
```

Algorithm A.15: Interpolate $J_T(x^+ + w)$)

```
1  Sidx,   Sidp   = id_from_m(x0,                    SiO2_min, SiO2_size,
       SiO2_len)
2  Caidx1, Caidp1 = id_from_m(x1/SiO2_bins[Sidx],    CaO_min,  CaO_size,
       CaO_len)
3  Caidx2, Caidp2 = id_from_m(x1/SiO2_bins[Sidx+1], CaO_min,  CaO_size,
       CaO_len)
4
5  ( (1 - Sidp) * ( (1 - Caidp1) * cost_T[Sidx   ,Caidx1   ,x2a,x3a]
6                   + Caidp1  * cost_T[Sidx   ,Caidx1+1,x2a,x3a] )
7       + Sidp * ( (1 - Caidp2) * cost_T[Sidx+1,Caidx2   ,x2a,x3a]
8                   + Caidp2  * cost_T[Sidx+1,Caidx2+1,x2a,x3a] ))
```

Algorithm A.16: J$_T$*calc*

```
1  def calc_JTmn_wrapper(cost_R, Xn, grid_args, bins, heights, n=30):
2
3      diff = -10        # initial value, can be any negative
4      prev_diff = -100 # initial value, negative, not = diff
5      gamma = 0.5       # discount factor in DP
6
7      # Calc. cost and controller in T-1
8      U_N, cost_T = cost_min_U(cost_R)
9      U_arr.append(U_N)        # store controller arrays in step T, T-1, ...
10     cost_arr.append(cost_T) # store cost arrays in step T,T-1,...
11
12     # Calculate DP problem n times
13     for i in range(n):
14         EJN = parallel_calc( E_JN,
15                              Xn[:,:,:,:,:,0],
16                              Xn[:,:,:,:,:,1],
17                              Xn[:,:,:,:,:,2],
18                              Xn[:,:,:,:,:,3],
19                              parallel=parallel)
20         U_N, cost_T = cost_min_U( cost_R + gamma * EJN )
21
22         U_arr.append(U_N)
23         cost_arr.append(cost_T)
24
25         # Calculate how many cells of the optimal control table have
               changed in this
```

```
26          # iteration
27          if i > 0:
28              diff = np.sum(np.abs(U_N - U_arr[-2]) > 0)
29              print( diff )
30              if prev_diff == 0 and diff == 0: # if none changed then stop
                    iterating
31                  break;
32              prev_diff = diff
33
34      # return U_arr[-1]
35      return U_arr, cost_arr
```

Algorithm A.17: JN

```
1  @vectorize(['float32(float32, float32, float32, float32)'], nopython =
       True)
2  def JN(x0,x1,x2,x3):
3      """ Calculates JT-n(x+,w)"""
4      def id_from_m(x0, x0_min, x0_size, x0_len):
5          x0id = (x0 - x0_min) / x0_size
6          x0idx = max( min( int(np.floor(x0id)), x0_len - 2), 0)
7          x0idp = x0id - x0idx
8          return x0idx, x0idp
9
10     Sidx,   Sidp   = id_from_m(x0, SiO2_min, SiO2_size, SiO2_len)
11     Caidx1, Caidp1 = id_from_m(x1 / SiO2_bins[Sidx],
12                                CaO_min, CaO_size, CaO_len)
13     Caidx2, Caidp2 = id_from_m(x1 / SiO2_bins[Sidx+1],
14                                CaO_min, CaO_size, CaO_len)
15
16     x2a = int(x2)
17     x3a = int(x3)
18
19     return ( (1 - Sidp) * ( (1 - Caidp1) * cost_T[Sidx   ,Caidx1   ,x2a,x3a
           ]
20                               + Caidp1   * cost_T[Sidx   ,Caidx1+1,x2a,x3a
                                     ] )
21                 + Sidp * ( (1 - Caidp2) * cost_T[Sidx+1,Caidx2   ,x2a,x3a
                       ]
22                               + Caidp2   * cost_T[Sidx+1,Caidx2+1,x2a,x3a
                                     ] ))
```

Algorithm A.18: EJN

```
1  @vectorize(['float32(float32, float32, float32, float32)'], nopython =
       True)
2  def E_JN(x0,x1,x2,x3):
3      """Calculates JT-n(x+) = E[JT-n(x+,w)] = Sum_i[JT-n(x+,wi)*p(wi)]"""
4      return np.dot( JN(x0+bins[:,0], x1+bins[:,1], x2+bins[:,2], x3+bins
           [:,3]), heights)
```

Algorithm A.19: cost_min_U

```
def cost_min_U(cost_U):
    """ This is the minimization for selecting the best U
        returns |indices of optimal U|,|cost of optimal U|
        Works on min 2D array, minimizes last dim."""
    sh=0
    if len(cost_U.shape) > 2: # if not 2D -> transform
        sh = cost_U.shape
        cost_U = cost_U.reshape((-1,sh[-1]))

    U_opt = np.argmin(cost_U, axis=1)
    cost_Tmn = cost_U[np.arange(cost_U.shape[0]), U_opt]

    if sh: # back transform
        U_opt = U_opt.reshape((sh[:-1]))
        cost_Tmn = cost_Tmn.reshape((sh[:-1]))

    return U_opt, cost_Tmn
```

Algorithm A.20: parallel$_c alc$

```
def parallel_calc(func, Var1, Var2, Var3=[], Var4=[], parallel=1):
    """
    f - function to parallellize,
    Var1 - valiable to parallelize by (x0)
    Var2 - x1
    """
    if parallel:
        def ff(func, r, out, Var1, Var2, Var3=[], Var4=[]):
            res = {'r':r}
            if len(Var3) == 0:
                res['C'] = func(Var1, Var2)
            else:
                res['C'] = func(Var1, Var2, Var3, Var4)
            out.put(res)

        first = True
        result=[]
        out_p = Queue()
        ranges = [x for x in np.array_split(range(Var1.shape[0]),
            cpu_count())
                  if x.size != 0]
        open_processes=[]

        for r in ranges:
            if len(Var3) == 0:
                x = Process(target = ff, args=(func, r, out_p, Var1[r],
                    Var2[r],) )
            else:
                x = Process(target = ff, args=(func, r, out_p, Var1[r],
                    Var2[r],
                                               Var3[r], Var4[r],) )
            open_processes.append(x)
```

```
30              x.start()
31
32          for process in open_processes:
33              temp = out_p.get()
34              if first:
35                  result = np.empty((np.hstack((Var1.shape[0], temp['C'].
                        shape[1:]))),
36                                    dtype=np.float32)
37                  first = False
38              result[temp['r']] = temp['C']
39
40          return result
41
42      else:
43          if len(Var3) == 0:
44              return func(Var1, Var2)
45          else:
46              return func(Var1, Var2, Var3, Var4)
```

Algorithm A.21: controller implementation

```
1  def controller(cont_tab, states, inputs, grids):
2      """Needed for unknown numba type inference reasons"""
3
4      def controller_wrapped(cont_table, states, inputs,
5                             SiO2_bins, SiO2_min, SiO2_size, SiO2_len,
                                  CaO_min, CaO_size,
6                             CaO_len, Ore_bins, Ore_min, Ore_size, Ore_len,
                                   Coal_min,
7                             Coal_size, Coal_len, Lime_min, Lime_size,
                                  Lime_len):
8          """ Apply controller """
9          def id_from_m(x0, x0_min, x0_size, x0_len):
10             """Creates index x0idx from an amount x0, and returns
                   distance measure x0idp"""
11             x0id = (x0 - x0_min) / x0_size
12             x0idx = max( min( int(np.floor(x0id)), x0_len - 2), 0)
13             x0idp = x0id - x0idx
14             return x0idx, x0idp
15
16         def InputN(x0,x1,x2,x3):
17             """ Interpolates U[sio2_id,cao_id,Ore,Coal], *id-s are
                   indexes already"""
18             Oidx,  Oidp  = id_from_m(x2, Ore_min, Ore_size, Ore_len)
19             Cidx1, Cidp1 = id_from_m(x3 - Ore_bins[Oidx]/1.83,
20                                      Coal_min, Coal_size, Coal_len)
21             Cidx2, Cidp2 = id_from_m(x3 - Ore_bins[Oidx+1]/1.83,
22                                      Coal_min, Coal_size, Coal_len)
23             x0a = int(x0)
24             x1a = int(x1)
25             return ( (1 - Oidp) * ( (1 - Cidp1) * cont_table[x0a, x1a,
                   Oidx,  Cidx1]
```

```
26                                          + Cidp1  * cont_table[x0a, x1a,
                                                Oidx,  Cidx1+1] )
27                          + Oidp * ( (1 - Cidp2) * cont_table[x0a, x1a,
                                Oidx+1,Cidx2]
28                                          + Cidp2  * cont_table[x0a, x1a,
                                                Oidx+1,Cidx2+1] ))
29
30          def UN(x0,x1,x2,x3):
31              """ Interpolates U[SiO2,Cao,-,-] -,- are not interpolated yet
                    """
32              Sidx,   Sidp   = id_from_m(x0, SiO2_min, SiO2_size, SiO2_len)
33              Caidx1, Caidp1 = id_from_m(x1 / SiO2_bins[Sidx],
34                                          CaO_min, CaO_size, CaO_len)
35              Caidx2, Caidp2 = id_from_m(x1 / SiO2_bins[Sidx+1],
36                                          CaO_min, CaO_size, CaO_len)
37
38              return ( (1 - Sidp) * ( (1 - Caidp1) * InputN(Sidx  ,Caidx1
                    ,x2,x3)
39                                          + Caidp1  * InputN(Sidx  ,Caidx1
                                                +1,x2,x3) )
40                          + Sidp * ( (1 - Caidp2) * InputN(Sidx+1,Caidx2
                                ,x2,x3)
41                                          + Caidp2  * InputN(Sidx+1,Caidx2
                                                +1,x2,x3) ) )
42
43          U_arr = []
44          for i in range(len(inputs)):
45              U_arr.append( UN(states[i,0], states[i,1], inputs[i,0],
                    inputs[i,1]) )
46
47          return  Lime_size * np.maximum( np.minimum(np.array(U_arr),
                Lime_len), Lime_min)
48
49      return controller_wrapped(cont_tab,
50                                states,
51                                inputs,
52                                grids['SiO2_prop']['bins'],
53                                grids['SiO2_prop']['min'],
54                                grids['SiO2_prop']['size'],
55                                grids['SiO2_prop']['len'],
56                                grids['CaO_prop']['min'],
57                                grids['CaO_prop']['size'],
58                                grids['CaO_prop']['len'],
59                                grids['Ore_prop']['bins'],
60                                grids['Ore_prop']['min'],
61                                grids['Ore_prop']['size'],
62                                grids['Ore_prop']['len'],
63                                grids['Coal_prop']['min'],
64                                grids['Coal_prop']['size'],
65                                grids['Coal_prop']['len'],
66                                grids['Lime_prop']['min'],
67                                grids['Lime_prop']['size'],
68                                grids['Lime_prop']['len'])
```

Bibliography

[1] R. Nightingale, R. Dippenaar, and W.-K. Lu, "Developments in blast furnace process control at port kembla," *Metallurgical and Materials Transactions B*, vol. 31, pp. 993–1003, 2000.

[2] A. Das, J. Maiti, and R. Banerjee, "Process control strategies for a steel making furnace using ann with bayesian regularization and anfis," *Expert Systems with Applications*, vol. 37, no. 2, pp. 1075–1085, 2010.

[3] V. M. Gasparini, L. F. A. de Castro, A. C. B. Quintas, V. E. de Souza Moreira, A. O. Viana, and D. H. B. Andrade, "Thermo-chemical model for blast furnace process control with the prediction of carbon consumption," *Journal of Materials Research and Technology*, vol. 6, no. 3, pp. 220–225, 2017.

[4] B. Vasu Murthy, Y. V. Pavan Kumar, and U. V. Ratna Kumari, "Fuzzy logic intelligent controlling concepts in industrial furnace temperature process control," in *2012 IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, pp. 353–358, 2012.

[5] V. Rybolovlev, A. Krasnobaev, N. Spirin, and V. Lavrov, "Principles of the development and introduction of an automated process control system for blast-furnace smelting at the magnitogorsk metallurgical combine," *Metallurgist*, vol. 59, 2015.

[6] Ismail Ekmekçi, Y. Yetisken, and Ünal Çamdali, "Mass balance modeling for electric arc furnace and ladle furnace system in steelmaking facility in turkey," *Journal of Iron and Steel Research, International*, vol. 14, no. 5, pp. 1–55, 2007.

[7] O. Y. Sheshukov, I. V. Nekrasov, A. V. Sivtsov, M. M. Tsimbalist, A. I. Stepanov, D. K. Egiazaryan, and V. V. Kataev, "Electric characteristic of steel-making electric furnace and the process control," in *Materials, Mechanical Engineering and Manufacture*, vol. 268 of *Applied Mechanics and Materials*, pp. 1376–1379, Trans Tech Publications Ltd, 2013.

[8] B. T.A, "A multilevel resource-saving blast furnace process control," in *Bulletin of the South Ural State University*, vol. 21 of *Computer Technologies, Automatic Control, Radio Electronics*, p. 136–146, 2021.

[9] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, and A. Juditsky, "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, vol. 31, no. 12, pp. 1691–1724, 1995.

[10] T. Bohlin, *Practical grey-box process identification.* Springer-Verlag London Limited, 2006.

[11] L. Ljung, *System Identification*, pp. 163–173. Boston, MA: Birkhäuser Boston, 1998.

[12] L. Ljung, "Prediction error estimation methods," *Circuits, Systems and Signal Processing*, vol. 21, no. 1, p. 11–21, 2002.

[13] I. J. Myung, "Tutorial on maximum likelihood estimation," *Journal of Mathematical Psychology*, vol. 47, no. 1, pp. 90–100, 2003.

[14] B. De Moor, P. Van Overschee, and W. Favoreel, *Algorithms for Subspace State-Space System Identification: An Overview*, pp. 247–311. Boston, MA: Birkhäuser Boston, 1999.

[15] H. Garnier, "Direct continuous-time approaches to system identification. overview and benefits for practical applications," *European Journal of Control*, vol. 24, pp. 50–62, 2015.

[16] P. M. P. Christodoulos A. Floudas, *Encyclopedia of Optimization.* Springer New York, NY, 2009.

[17] A. Ruhe, "Accelerated gauss-newton algorithms for nonlinear least squares problems," *BIT Numerical Mathematics*, vol. 19, no. 3, pp. 356–367, 1979.

[18] K. Levenberg, "A method for the solution of certain problems in least squares," *Quart. Appl. Math.*, vol. 2, p. 164–168, 1944.

[19] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math.*, vol. 11, p. 431–441, 1963.

[20] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.

[21] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the monte carlo method is so important today," *WIREs Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014.

[22] S. Bandyopadhyay, S. Saha, U. Maulik, and K. Deb, "A simulated annealing-based multiobjective optimization algorithm: Amosa," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 3, pp. 269–283, 2008.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[24] S. Bhojanapalli, B. Neyshabur, and N. Srebro, "Global optimality of local search for low rank matrix recovery," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[25] D. P. Bertsekas, *Neuro-dynamic programmingNeuro-Dynamic Programming*, pp. 2555–2560. Boston, MA: Springer US, 2009.

[26] G. Infanger, "Chapter 5 - dynamic asset allocation strategies using a stochastic dynamic programming aproach," in *Handbook of Asset and Liability Management* (S. Zenios and W. Ziemba, eds.), pp. 199–251, San Diego: North-Holland, 2008.

[27] J. Rust, "Chapter 14 numerical dynamic programming in economics," vol. 1 of *Handbook of Computational Economics*, pp. 619–729, Elsevier, 1996.

[28] L. Marescot, G. Chapron, I. Chadès, P. L. Fackler, C. Duchamp, E. Marboutin, and O. Gimenez, "Complex decisions made simple: a primer on stochastic dynamic programming," *Methods in Ecology and Evolution*, vol. 4, no. 9, pp. 872–884, 2013.

[29] J. M. Hutchinson and J. M. McNamara, "Ways to test stochastic dynamic programming models empirically," *Animal Behaviour*, vol. 59, no. 4, pp. 665–676, 2000.

[30] A. Gjelsvik, B. Mo, and A. Haugstad, *Long- and Medium-term Operations Planning and Stochastic Modelling in Hydro-dominated Power Systems Based on Stochastic Dual Dynamic Programming*, pp. 33–55. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[31] S. Yakowitz, "Dynamic programming applications in water resources," *Water Resources Research*, vol. 18, no. 4, pp. 673–696, 1982.

[32] J. N. Hooker, "Decision diagrams and dynamic programming," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (C. Gomes and M. Sellmann, eds.), (Berlin, Heidelberg), pp. 94–110, Springer Berlin Heidelberg, 2013.

[33] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[34] M. Sniedovich, *Dynamic Programming: Foundations and Principles, Second Edition*. CRC Press., 2010.

[35] R. Bellman, "The theory of dynamic programming," *Bull. Amer. Math. Soc.*, vol. 60, pp. 503–515, 1954.

[36] J. Rust, "Using randomization to break the curse of dimensionality.," *Econometrica*, vol. 65, no. 3, pp. 485–516, 1997.