Università degli Studi di Padova

Università degli Studi di Padova

Department of Information Engineering

Bachelor Thesis in
Information Engineering

PID control of electrodes movements in submerged arc furnaces

Supervisor:
Prof. Damiano Varagnolo
Università di Padova

Bachelor candidate:
Enrico Miotto
1216484

Co-Supervisor:
Manuel Sparta
NORCE Norwegian Research Centre

Academic Year 2021/2022
Graduation date: July 20, 2022

≪"Unglücklich das Land, das keine Helden hat!" [...]
"Nein. Unglücklich das Land, das Helden nötig hat."≫

(Bertolt Brecht, Lebens des Galilei)

Abstract

One of the most widely employed control technique is PID - i.e., Proportional, Integrative, Derivative control. Its popularity is due to its simplicity, robustness, and being a model-free approach, which allow it to be applied to nonlinear systems for which model-based control, requiring a system model with good generalization capabilities, may be a too complicated task. Summarizing, if obtaining a good model is difficult, the PID model-free control strategy may be a viable one.

However finding suitable PID settings so to ensure good closed loop control performance by experimenting on a physical system could be a daunting task (especially if the plant costs hundreds of millions of Euro). Employing a simulator as a substitute of the physical system for this task may then be useful; i.e., one may compare the results based on real measures and the simulator, tune the controller on the simulator, and have some expectations of the behaviour of the controlled system once from the simulation realm one moves to the physical one.

In this thesis we thus consider the situation above, i.e., starting from finite elements and finite volumes simulators (two of the most common approaches to modelling a system) to tune PID controllers that will after be used in real life. However, both approaches may be computationally and software demanding, and trial and error tuning of PIDs based on such models may take too much time.

The approach we chose on this Thesis is then to exploit meta models, i.e., models of models that have been obtained by opportune statistical analyses of the data obtainable from the physics-based models above, and whose computational requirements are only a fraction of the original ones, trading though off accuracy of the results (in the sense of how well the meta models model the original systems).

The goal of this thesis is then to show how one may implement and tune simple PID controller based on meta models that simulate the electrical conditions in submerged arc furnaces. In this situation the controller acts on the electrode's position in order to keep constant an output value (such as electrode's resistance, power etc.). We thus test the approach on this specific system, and focus on comparing (and

understanding) the results that one obtains with specific process noise based on Perlin Noise (a specific type of dynamic disturbance that accurately represents the smooth variations typical of furnaces inputs during the day).

We discovered that such a controller is effective when it has to handle a limited number of noisy inputs. If noise is to be applied to all input variables, it is better to turn to more advanced control techniques. On the other hand, the simplicity and efficiency of this implementation allow it to be versatile and effective in many cases. A final important aspect that has been observed is the sizing of the time interval that the controller uses to scan the various steps. Despite the objective difficulties in determining it accurately, through a number of simulations we were able to estimate it with a good approximation, which allows us to have both a clean control and a good calculation speed.

This work has been developed in collaboration with NORCE - Norwegian Research Centre as part of their project SAFECI, "Electrical Conditions in Submerged Arc Furnaces – Identification and Improvement", with financial support from The Research Council of Norway (Project number 326802) and the companies Elkem, Eramet Norway, Finnfjord, and Wacker Chemicals Norway.

Contents

Chapter 1

Brief introduction to PID

1.1 Introduction

First, the theory concerning PID controllers will be introduced from an intuitive point of view. More mathematical details can be found in other treatises (see [1] [9]), but for this Thesis everything concerning PID controllers applied to linear systems is irrelevant. Submerged arc furnaces are highly non-linear and complex systems. In such cases, the PID control is calibrated by means of simulations or techniques that are outside the scope of this discussion. For this reason, we will focus mainly on the heuristic idea behind PID control, avoiding mathematical overload.

The PID controller is a control algorithm with a predefined structure, which is calibrated by changing the value of certain parameters. Due to its simplicity of use, combined with appreciable effectiveness in various fields of application, it is by far the most widely used control algorithm in industrial applications. Mathematically speaking, the PID controller is a dynamic system that processes an input signal, called "error" (obtained as the difference between the reference and the controlled variable e(t) = r(t) - y(t)) then giving as output a control signal u(t).

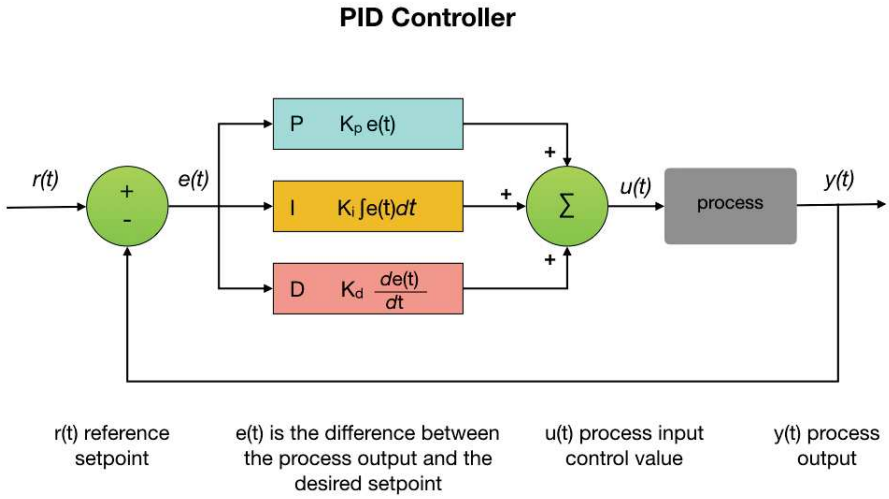


Figure 1.1: PID block diagram

The success of PIDs is mainly due to:

- Remarkable effectiveness in regulating a wide range of industrial processes (thermal, mechanical, etc.);
- Possibility of being made with different techniques (mechanical, hydraulic, electronic both analog and digital, etc.);
- Relative ease of calibration of working parameters since their use does not require from the operator a detailed knowledge of the process to be controlled;
- Good effectiveness/cost ratio because in the presence of unavoidable noise conditions or inappropriate calibration errors, the effectiveness of the control law decays, so the use of other, more refined control systems would be useless, as their performance would become comparable to that of PIDs;
- Ductility in their use; in fact, they are often used to achieve more articulated configurations to meet complex requirements.

The PID controller in its basic form, has a structure comprising the sum of three control terms

$u(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(\tau)d\tau + K_D \cdot \frac{de(t)}{dt}$

The parameters that identify the PID therefore are KP, KI and KD and they are called also controller degrees of freedom.

Below are brief explanations of the role of each of the three components of the PID. For more details, please refer to [1] and [2].

1.1.1 Proportional action

The effect of proportional action depends on the error computed by the difference between the setpoint value and the current output value. If the error grows, then the proportional action gives a stronger contribution to reduce the entity of it. An higher value of Kp increases the speed of the system, but also the oscillations.

It is important to remember that there is always a steady state error in proportional control. It decreases when Kp is augmented, but the oscillations increase too.

1.1.2 Integral action

The Integral action depends on the value of error's integral. The main characteristic of this part of the control is that it eliminates the steady state error given by the Proportional part. Integral term is increased if the error is positive and decreases if the error is negative. For that reason, it could explode if the actuators have limits on their action. This phenomena is known as "windup" and it will be discussed more in detail in the next section.

1.1.3 Derivative action

The last action is the Derivative one. Its value is computed using the error's derivative and an opportune constant Kd.

It is sensitive to rapid changes in error, and tries to anticipate them. The effect is a reduction in signal fluctuations around the setpoint, while the offset is not affected.

It is necessary to be careful with calibration, because this effect could make the system unstable. Often in controllers Kd is very small or completely absent, especially if the system to be controlled is not subject to sudden changes.

1.2 Windup

Windup is a non linear phenomenon caused by the interaction of integral action and saturation. When the actuators have limitations, it may happen that the control variable reaches the actuator limits. In this way the feedback loop is broken and the system runs as an open loop because the actuator will remain at its limit. If a controller uses the integrating action, the error will continue to

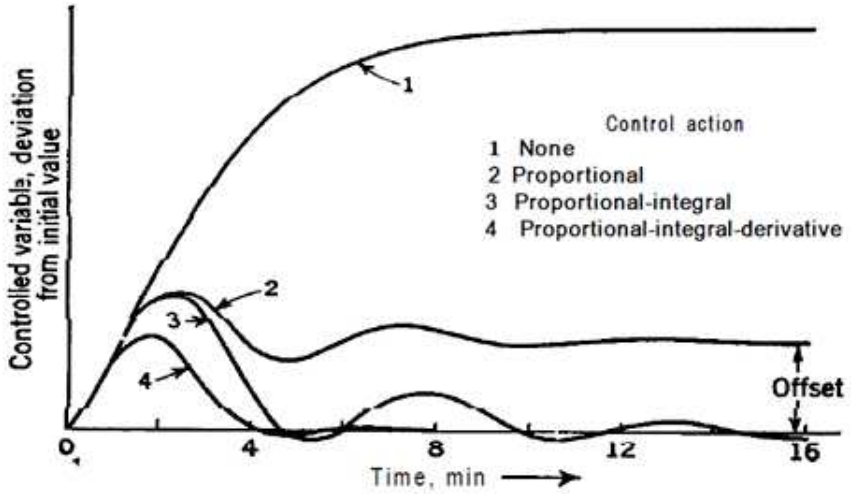


Figure 1.2: An example of the three actions of a PID controller

be integrated. The integral term may become very large (it "winds up"). After that, things return normal only if the error has opposite sign for a long period.

Consequently there may be large transients when the actuator saturates.

1.2.1 Avoid windup

One of the simplest techniques which allow to handle windup is stopping the integral action whenever the output saturates. This approach is similar to that followed in the early phases of feedback control. The integral action was integrated with the actuator, by having a motor drive the valve directly. Integration stopped when the valve stopped. Procedures that implement this scenario are called Incremental Algorithms.

Many alternative methods are used in order to limit this bad effect, such as Setpoint Limitation, which introduces limits on the setpoint variations, so the controller output never reaches the actuator limits. The windup caused by noise is not avoided by this choice. The last method, which is implemented in the controller considered in this Thesis, is Back-Calculation and Tracking.

When the output saturates, the integral term in the controller is recomputed so that its new value gives an output at the saturation limit. In this way, the integral term growth is bounded and can not explode.

1.3 Pseudo code

PIDs are simple and intuitive. The following pseudo-code shows principal operations required to implement such controllers. Note that the integral component is not allowed to surpass the limit and it is equal to the maximum output accepted when there is saturation. The output is bounded too.

Algorithm 1 PID algorithm

Input: setpoint, measured_value, dt, max_out
Output: output

$last\_error \leftarrow 0$
$integral \leftarrow 0$
start:
$error \leftarrow setpoint - measured\_value$
$integral \leftarrow integral + error * dt$
if $integral > max\_out$ **then**
    $integral \leftarrow max\_out$
end if
$derivative \leftarrow (error - last\_error)/dt$
$output \leftarrow Kp * error + Ki * integral + Kd * derivative$
if $output > max\_out$ **then**
    $output \leftarrow max\_out$
end if
$last\_error \leftarrow error$
$wait(dt)$
$goto$: start

Chapter 2

Submerged arc furnaces simulator

2.1 Introduction

The online simulator considered in this thesis, [7], was developed by NORCE and implements a metamodel for submerged arc furnaces, based on the statistical analysis of the results of a physics-based model (e.g., based on the finite element method). Once the parameters of interest have been identified from the finite element model and an input and output database produced from it, a surrogate linear model is produced using Partial Least Square Regression, PLSR. Some details about the system considered and the parameters of interest are given below.

2.2 The furnace

An example of the furnace taken as a reference in the development of the simulator. The diameter is 10.5 metres and the height 5.8 metres. The electrodes have a diameter of 1.55 metres and are arranged in an equilateral triangle with a distance between the centres of 3.8 metres. The electrodes have the ability to move vertically. A change in their position results in a change in the behaviour of the furnace because the resistance of the current paths in the furnace is modified. The controller developed for this thesis acts on this parameter to control the system. The behaviour of the system is very complex, which is why a simple PID controller whose constants can be adjusted through repeated simulations. The main aim of this Thesis is to provide an easy-to-use and reliable controller that allows multiple tests to be performed based on the behaviour detected through the metamodel. With such data, comparisons can eventually be made with ex-

perimental measurements to ascertain the reliability of the simulations. For more details on the parameters used for the simulator, please refer to [8].

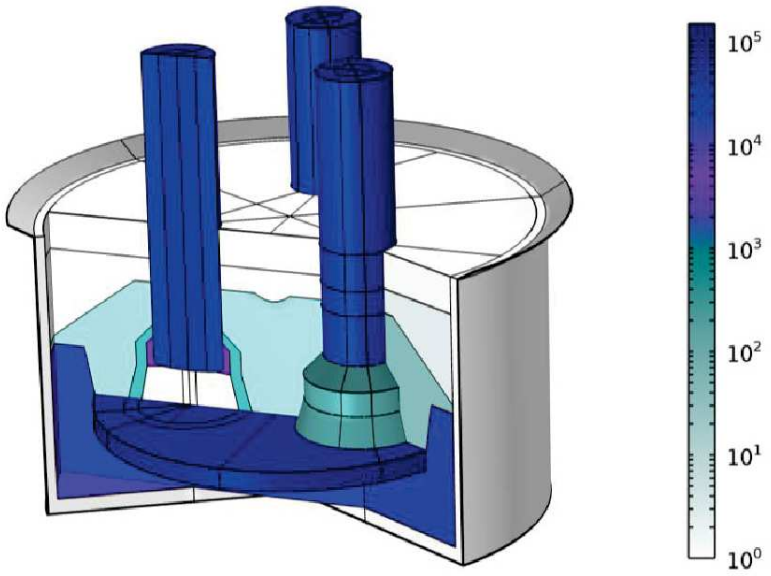


Figure 2.1: Submerged arc furnace scheme [8].
The colors encode the resistivity of the different zones.

2.3 Input Output variables

The various parameters considered as inputs but on which the controller cannot act on:

- I1: current in the first electrode [kA];
- I2: current in the second electrode [kA];
- I3: current in the third electrode [kA];
- CW_T: Crater wall thickness [cm];
- σ_cw: crater wall conductivity [S/m];
- σ_SiC12: conductivity of the SiC banks separating electrodes 1-2 [S/m];
- σ_SiC23: conductivity of the SiC banks separating electrodes 2-3 [S/m];
- σ_SiC31: conductivity of the SiC banks separating electrodes 3-1 [S/m];

- σ_arc: the arc in the crater cavity is modelled as a resistive element, whose conductivity is the same of the ionized atmosphere equal to 7000 S/m.

The system inputs on which the controller has freedom to act are:

- z1: position of the first electrode [m];
- z2: position of the second electrode [m];
- z3: position of the third electrode [m];

The list of main outputs is as follows. The controller can be called upon to maintain a certain value of any of these parameters, as will be explained in detail below.

- P_tot: total active power [MW];
- Q_tot: total reactive power [MVAR];
- R_tot: total resistance [mΩ];
- X_tot: total reactance [mΩ];
- cosphi: power factor;
- Psh_tot: steel shell power [MW];
- P_roof: steel roof power [MW];
- Pcw_tot: total power crater walls [MW];
- Psic_tot: total power SiC [MW];
- Phc_tot: total power hot charge [MW];
- Pcc_tot: total power cold charge [MW];
- Psic_12: power SiC electrode 1-2 [MW];
- Psic_23: power SiC electrode 2-3 [MW];
- Psic_31: power SiC electrode 3-1 [MW];

Finally, some local outputs, calculated for each electrode, are considered. This makes it possible, with three controllers, to act locally on the electrodes to keep certain outputs constant, which can also be chosen differently from electrode to electrode.

- electrode's active power [MW];
- electrode's reactive power [MVAR];
- electrode's resistance [mΩ];
- electrode's reactance [mΩ];
- electrode's voltage [V];
- electrode's power crater [MW];
- electrode's power crater walls [MW];

2.4 Meta Model code's structure

The meta-model simulator is organised in the following files:

- `MainMetaModel.py`
- `MetaModelClass.py`
- `NoiseFunctions.py`
- `PlotFunctions_Metamodel.py`

These files were created and designed by Mads Fromreide and provided to be adapted to the present discussion. Any modifications made will be listed below and their function will be outlined in broad terms.

2.4.1 MainMetaModel.py

MainMetaModel.py is the executable file, which allows simulating the operation of the furnace and choosing which input quantities will be noisy. No significant changes have been made to it.

2.4.2 MetaModelClass.py

MetaModelClass.py is the class that constitutes the actual simulator. It allows a metaModel object to be created and provides the `Simulate` method that simulates the operation of the furnace, based on the meta-model, in a given time frame. In this file, certain parameters of interest can be set, such as the limits of the

electrode positions, the setpoints of the output quantities to be controlled, the maximum speed of the electrodes, the initial values of the currents, electrode positions and the various inputs, and the amplitudes of the various errors on the inputs. The time interval and the value of dt are provided as input to the constructor of the class and can then be inserted into the executable file in which the metaModel object is created. It is also possible to set a Boolean flag to true to make the controller intervene during the execution of the `Simulate` method.

2.4.3 NoiseFunction.py

NoiseFunction.py contains the functions that manage the creation of noise and the operation of the controller that is the subject of this Thesis. This source file has been heavily modified in order to include all the necessary functionality. The `controller\_function` is designed to be called at each iteration of the loop in the Simulate function of MetaModelClass.py. Its features will be explained in more detail in Chapter 3. It relies on a number of other minor functions that handle the various issues that may arise during control. The `Noise` function deals with modelling certain types of noise to be applied to the various input quantities. Mainly in this discussion we will focus more on Perlin Noise and Fractal Noise, which will be discussed in more detail in Chapter 4.

2.4.4 PlotFunctions_Metamodel.py

PlotFunctions_Metamodel.py consists of a number of useful functions for printing and graphing the various quantities involved such as currents, powers, etc. The only changes made concern the addition of functions that manage the plotting of certain outputs not previously considered.

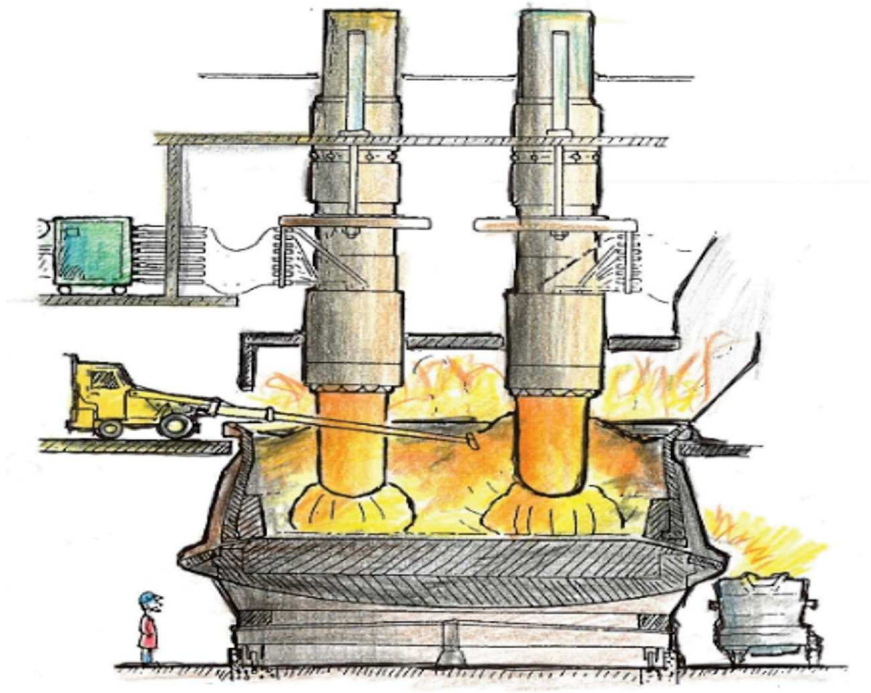


Figure 2.2: Drawing showing a submerged arc furnace.
The Si process - Drawings, by T. Hannesson, 2016, Elkem Iceland.

Chapter 3

The Controller

3.1 Introduction

The controller studied in this thesis is of the PID type. It has been implemented in the Python language, following the algorithm already outlined in Chapter 1. First its input/output specification will be exposed, the operation will be discussed later. The function signature that simulates the controller's behaviour is as follows:

```
controller_function(output_type,
setpoint, last_value_out, z, dt, integral, lastError,
maxSpeed, zmin, zmax, kp, Ki, Kd).
```

The function accepts the following mandatory inputs:

- output_type: the type of output needed (e.g. power, resistance, etc.);
- setpoint: setpoint that the PID will try to achieve;
- last_value_out: last measured output value;
- z: last position of the electrode [m];
- dt: time interval between two passes [s];
- integral: integral of the past error;
- lastError: last value of the error;
- maxSpeed: maximum speed which the electrode can reach [cm/h];

- zmin, zmax: minimum/maximum height [m];
- Kp, Ki, Kd: proportional/integral/derivative gain.

The function returns three values, one of which is the required output (in the case of this controller it is the new position of the electrode being considered), the other two are used to update the integral component and the measured error, they are therefore numerical values required to keep the controller up-to-date during its operation.

- z + dh: updated position of the electrode;
- integral: updated value of error's integral;
- lastError: updated value of the error for the next iteration.

3.2 Controller Operation

First, the controller calculates the error between the setpoint value and the last measured value of the output to be controlled. It is necessary that the integral term and the error value have already been initialised in order to be able to perform all necessary operations. For this reason, these values must be set to zero before starting a simulation. In the case of this Thesis, it was decided to initialise the controller parameters within the `Simulate()` function, in the MetaModel.py source file, which contains the class that deals with the operation of the meta-model.

3.2.1 Integral term and windup

The integral term is to be initialised outside the controller function, and is to be kept up-to-date using the second output of the function. Once the error has been calculated, the integral term is updated and an anti-windup check is performed using the function:

`clamp(value, lower, upper)`.

This function checks whether a given value has exceeded certain limits given as an argument. The integral component, even if the position of the controlled electrode reaches the limit given by zmax or zmin, continues to increase or decrease if not handled. In fact, it depends on the measured error which, when

the controller saturates, continues to be calculated and may increase even though the limits have already been reached. This phenomenon, as explained in the first chapter, is known as windup. In the present implementation of the controller, this has been remedied firstly by calculating the maximum allowed height variation, called dh_max.

This value also depends on the maximum permitted electrode speed, which in the present discussion is 75 cm/h.

Through the value dh_max, it is possible to calculate the contribution of the integrative part and ensure that it is never greater in absolute value than dh_max. By doing so, the integral term will stop growing once a certain limit is reached and the windup phenomenon will be very limited.

In order to obtain this result, the function clamp is invoked as follows:

```
integral += Ki*error*dt
integral = clamp(integral, -dh_max, dh_max).
```

It can be seen that the clamp function not only checks that the integral value is contained within the bounds, but also returns the correct value if it exceeds them, i.e. dh_max or -dh_max.

3.2.2 Maximum and minimum electrode's height management

It is important to note the presence of limits necessary for the correct operation of the controller. As mentioned earlier, electrodes can reach a maximum and minimum height, which are 0.10 m and 0.75 m respectively. These limits are respected through the intervention of the function:

```
isPositionValid(z,dh,zmin,zmax).
```

The returned value is a boolean, which thus makes it possible to determine whether the electrode height limits have been exceeded. If this is the case, the height control will be interrupted and the height will remain fixed at the maximum or minimum limit. This 'lock' will only intervene if the controller, in addition to having reached the limit, calculates an increase or decrease in height such that this limit is further exceeded. In the event that it attempts to move back to values within the limits, the position will not be locked but will be allowed to move to valid values.

Once the various checks on the windup and electrode positions have been carried out, the clamp function is invoked one last time to ensure that the maximum permitted speed is not exceeded. Finally, the value of lastError is updated with

the error just calculated and the three required outputs are returned.

3.3 Time Interval design

An input that greatly affects the performance of the controller is the time interval value dt. A smaller value leads to greater accuracy and speed of response, but also to a higher cost in terms of computation. It is not easy to estimate the optimal value of this parameter. For linear systems, whose time constants are known, it must be 1/20th of the slowest mode. The furnace considered in this Thesis is strongly non-linear, so some tests were preferred to find a trade-off between computational complexity and controller efficiency.

As a first simulation, we focused on the first of the three electrodes. The system was simulated for a number of iterations equivalent to 24 hours of operation, in the absence of noise on all inputs. The output to be controlled chosen was the electrode resistance, with a setpoint of 0.7 mΩ. Only the value of dt was varied, from 10 s to 90 s. All other parameters were kept constant at intermediate values between those possible in the online simulator. As limits, zmin = 0.1 m, zmax = 0.75 m and as maximum electrode speed 75 cm/h were set. The simulation results are shown in the figure 3.1. A loss of performance of the controller can clearly be seen, albeit very small.

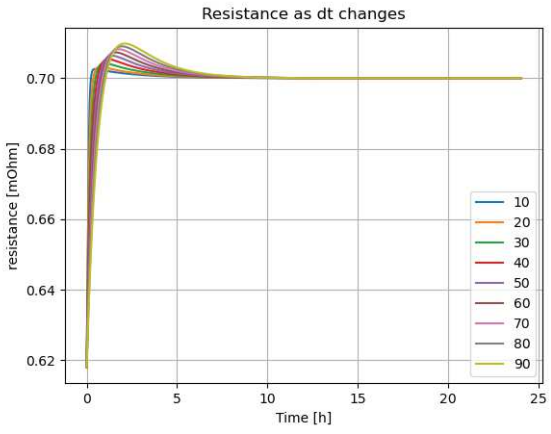
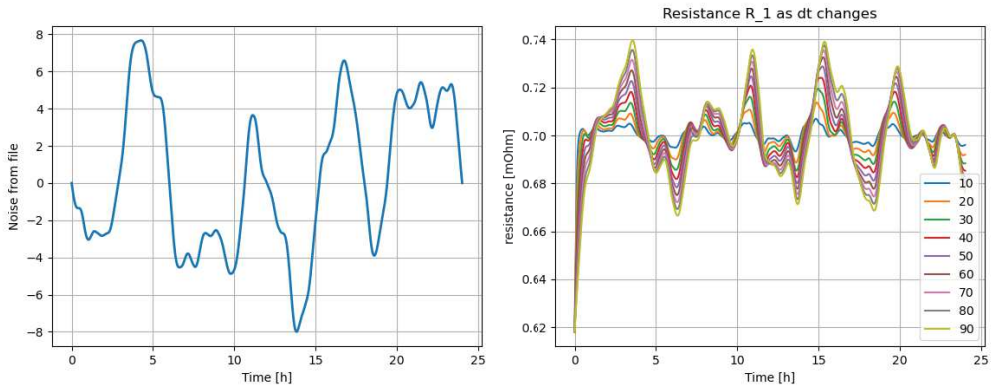


Figure 3.1: dt variation without noise

As a second step, noise was introduced on the first electrode current and the test was repeated. The noise used is Fractal Noise, the way in which it was generated will be explained in Chapter 4 of this discussion. The maximum amplitude of this noise in the simulation was 8 kA. This time, the difference in

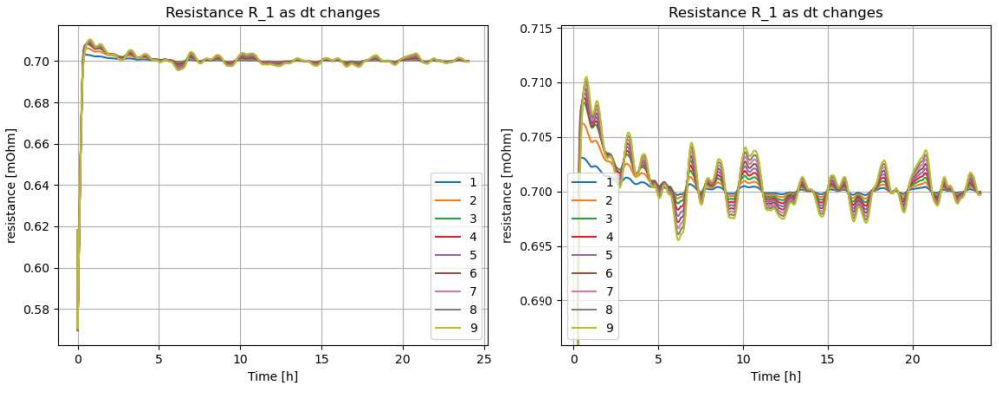
performance as dt increases was much more visible. Both the noisy current used and the result with regard to resistance control can be seen in figure 3.2.



(a) (b)

Figure 3.2: dt variation with noisy current

The same noise was maintained and this time dt was made to vary between 1 s and 9 s. Below is the result of the processing, and a zoom to observe the differences in more detail. It can be seen that the performance is better than the previous simulation, but that above a certain value the improvement starts to be reduced. It is also useful to point out that a small value of dt corresponds to a large computation time, due to the greater number of iterations performed. It should also be anticipated that, as constructed, Fractal Noise begins to be computationally expensive the larger the number of samples required. This aspect will be dealt with in more detail later on.



(a) (b)

Figure 3.3: Smaller values of dt

In conclusion, it can be seen that the performance of the controller starts to be very poor for dt values greater than 10 seconds. It must be considered that

noise can actually be added to all the system input parameters listed in Chapter 2. By doing so, performance may be even worse than in previous simulations, where the only noisy parameter was current. On the other hand, it is not advisable to keep the value of dt too low, given the high number of iterations for which the simulator was designed. For this reason, a dt value of around 10 s was considered a good compromise.

3.4 Further Improvements

Many things can still be improved. For example, the controller uses the position of the furnace electrodes as input and output. Instead, the quantities to be controlled and kept constant can be chosen from the outputs listed in Chapter 2. The quantity controlled and the quantity to be controlled do not therefore belong to the same domain. For this reason, it would be advisable to provide the controller with an error mapping, which allows different domains to talk to each other. This would also help in view of the fact that the electrodes saturate in position, and thus to achieve a more proportionate control by mapping the position range with the range possessed by the output variable. This would limit the saturation phenomenon.

Finally, when the noisy inputs become multiple, the controller often incurs the phenomenon of position saturation. This is due to the fact that the only parameter on which it can act is the height of the electrodes, and this is often not sufficient to properly control the behaviour of the furnace. It is therefore optimal to switch to a higher control level when saturation of the electrode positions occurs, thus allowing more sophisticated controllers to act and returning control to the PID once the saturation effect is extinguished.

In the industrial plant, this is addressed by changing the tapping position on the transformers that feed currents into the systems. This is not implemented in the current simulator and will not be discussed further.

Chapter 4

Noise

4.1 Introduction

The controller illustrated above is also designed to work in the event that noise affects the input parameters of the furnace during its operation. Consequently, it was decided to use a particular type of noise, which is very common in image processing due to its characteristics of regularity and apparent randomness, despite being a completely deterministic noise. Such noise is called Perlin Noise [4, 5] and was proposed by Ken Perlin in 1985, after which it has become increasingly popular in many engineering, computer and artistic applications.

4.2 Perlin Noise

Perlin Noise, as already mentioned, is very popular for image processing and the development of multimedia applications. Its characteristic is to simulate very well elements found in nature such as rough surfaces, depressions, flames, mists or sea surfaces. It owes this to its very regular appearance that very faithfully recreates what happens in the real world. Considering a rock surface, for example, a distant observer can detect depressions and roughness, but if the observer approaches two infinitesimal points they will be very close to each other and at an almost identical level. Consequently, when large overall variations but almost zero variations in an if concentrated in a short space/time interval are needed, adopting Perlin Noise can be a useful expedient.

4.2.1 Usage

Perlin Noise can be created in various dimensions. For example, using two dimensions, it can be used to recreate smoke or a rough surface. In three dimensions, it is widely used to reproduce landscapes full of depressions and reliefs, or wave-like motions. The case that is of interest in this discussion is in one dimension.

Figure 4.1: Example of three-dimensions Perlin Noise [4]

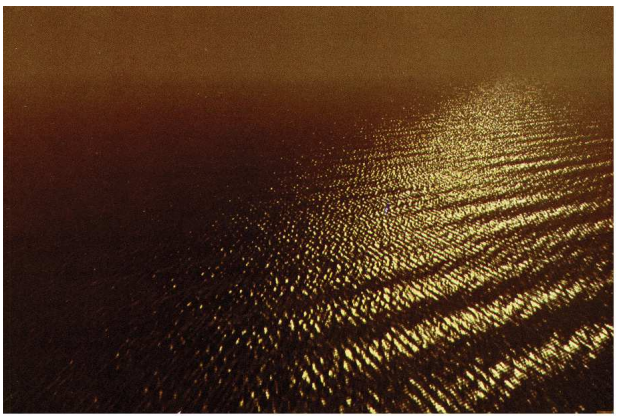


Figure 4.2: Example of three-dimensions Perlin Noise [4]

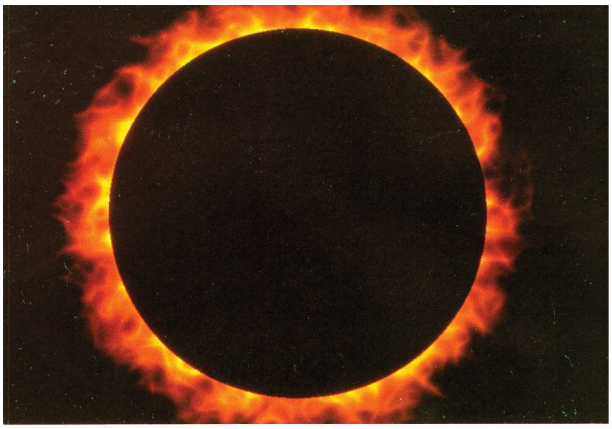


Figure 4.3: Example of two-dimensions Perlin Noise [4]

The aim was to add noise to the input quantities to the furnace, listed in Chapter 2. These quantities are one-dimensional signals, functions of time and considered constant. For example, the current flowing in the electrodes is an input having a certain value which is changed neither by the controller nor by external agents, and which is subject to noise within a certain range and which is desired to be as random as possible but at the same time very smooth.

A dedicated python library named perlin-noise [6] was used, which allows various parameters of the noise to be set, such as the number of octaves or the starting seed of the noise. The function that will be illustrated below makes it possible to create a noisy signal with a maximum amplitude and an indicative 'frequency' that is predetermined. An example of a noisy signal created in this manner can be seen below.

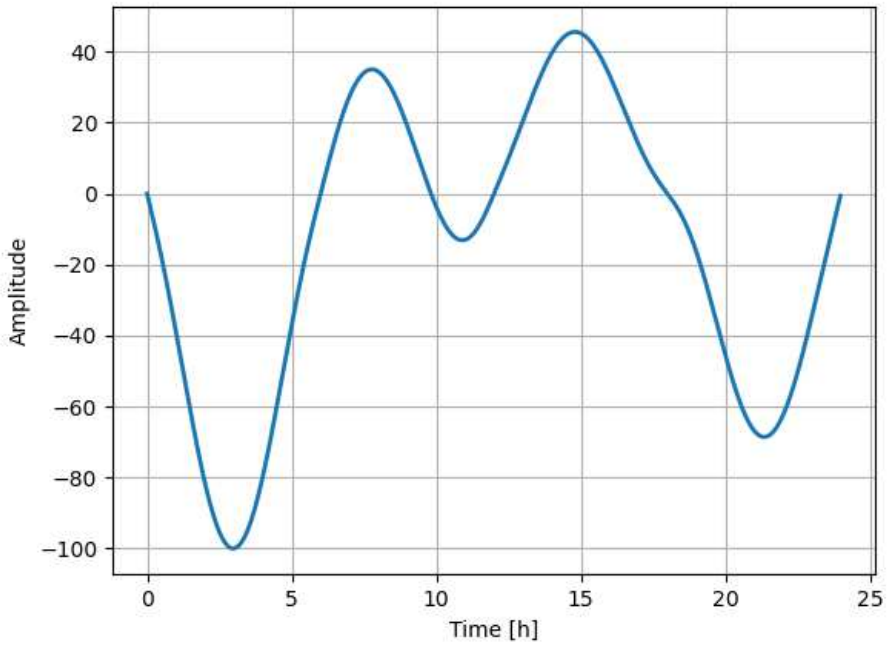


Figure 4.4: One dimensional Perlin Noise

It can clearly be seen that the noise thus obtained has the characteristics listed above, but is still not optimal for modelling the physical noise to which the various input parameters of the furnace are subjected. The main problem concerns the handling of octaves, which will be discussed in more detail in the next subsection. Evident is the presence of zeros whose positions divide the time axis into equal parts. Such a systematic nature of this phenomenon undermines the naturalness one wishes to achieve. Moreover, the perturbations are in any case

too regular and oscillatory in form. For this reason it was decided to implement Fractal Noise, based on Perlin Noise but with certain peculiarities. It will be explained in section 4.3.

4.2.2 Gradients

Perlin Noise is also called Gradient Noise, as it relies on the use of random gradients to be calculated.

In the present discussion, we will focus mainly on one- and two-dimensional Perlin Noise, which can easily be extended to the three- and multi-dimensional case.

The main parameter to be considered is the number of octaves. This value represents the number of units into which the space containing the noise is subdivided, and is therefore equivalent to the number of intervals into which the time axis is subdivided in the one-dimensional case treated in this thesis. In two dimensions, on the other hand, the number of octaves is the number of intervals into which the two main axes of the two-dimensional space into which the Perlin Noise is to be drawn are divided.

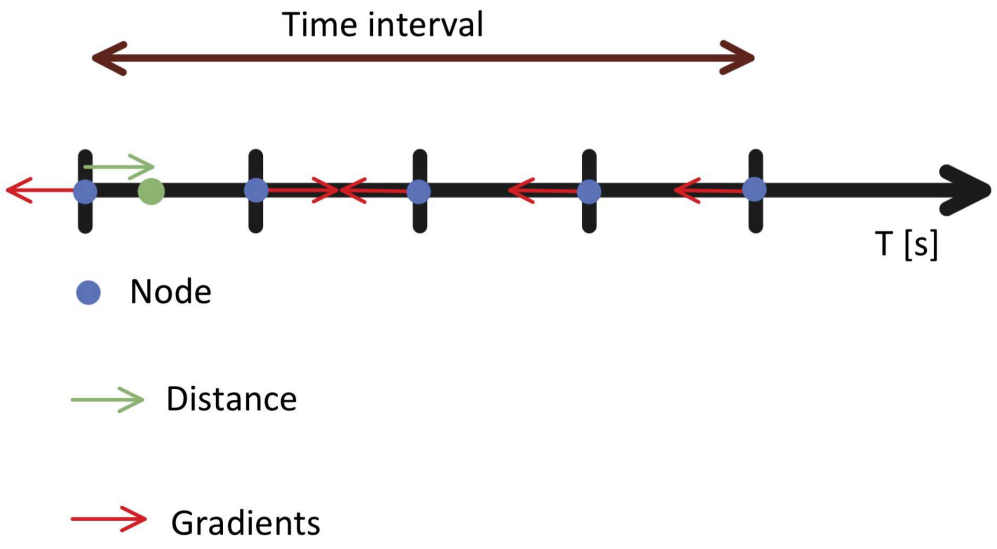


Figure 4.5: One dimensional Perlin Noise octaves and gradients

Once the number of octaves and thus the grid has been chosen, a random gradient of unit length is associated with each of its nodes. In the latest versions of Perlin Noise, this gradient is chosen from a defined number of possibilities instead of the infinite number of unit gradients that can be associated with a point. In the one-dimensional case, gradients can only have one direction coinciding with the one axis, the temporal axis, and two directions, i.e. to the right and to the left. Then for each point that does not belong to the grid nodes, the nearest node is considered and the scalar product between the distance from it and its gradient is calculated. Once all the scalar products have been calculated, the results are interpolated with a polynomial function of appropriate degree.

Once this procedure is understood, it can be deduced that the presence of zeros dividing the time axis into equal parts is due to the fact that at grid nodes, the distance to the nearest node is zero. The scalar product therefore results in zero. Furthermore, the case in which I have the minimum possible value is when the gradients are outside the distances of the considered interval, while I have the maximum value if the gradients are directed inwards.

4.3 Fractal Noise

The noise that will be described in this section is widely used, in one dimension, to recreate the tremors of an earthquake detected by a seismograph, or the random stroke of a pen on a sheet of paper. It is appreciated for its naturalness and apparent randomness, and is created from Perlin Noise, the characteristics of which were discussed in the previous section.

Fractal noise is noise that can be obtained by adding different noise components created with Perlin Noise. These components differ in the number of octaves and amplitude. The larger amplitude component has fewer octaves, and behaves as if it were the carrier. The other components have an increasing number of octaves as the amplitude decreases. In this way, irregularities with a higher 'frequency' will have a smaller modulus, while the more regular ones will make up the bulk of the noise amplitude. An example of fractal noise is the one in the image 4.6, obtained by summing 4 components of Perlin Noise with a duration of 24 h of number of octaves 4, 8, 16, 32 respectively and amplitudes 1, 0.5, 0.25 and 0.125 and redeemed to 8.

This type of noise is better than Perlin Noise for modelling disturbances in submerged arc furnaces. The rises and falls are very unpredictable, but at the

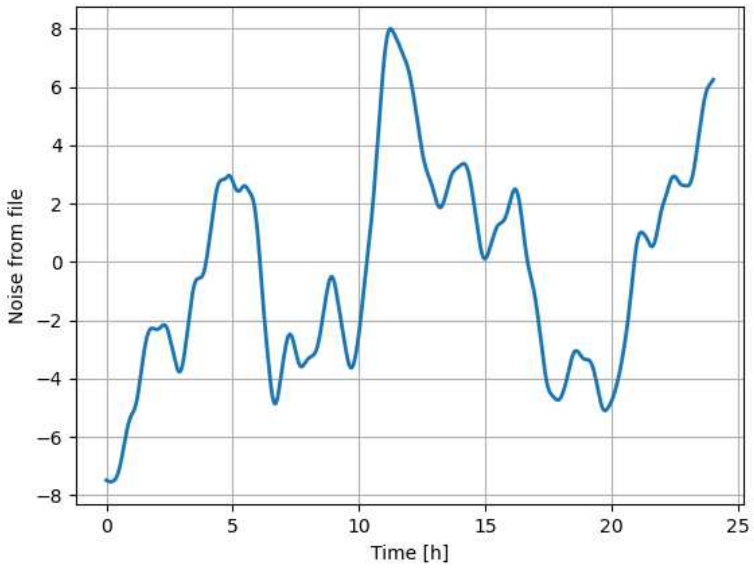


Figure 4.6: One dimensional Fractal Noise

same time gradual and irregular. The only shortcomings are of a computational nature. In fact, more Perlin Noise is required to create this noise, which translates into a greater complexity the more components are required. In the present discussion, we have chosen to keep 4 components as in the example image 4.6 scaled in the same way. The number of octaves of the carrier can instead be chosen by the user, as can the maximum amplitude of the noise. More details on the implementation can be found in the next section.

4.4 Implementation

In this section, the noise implementation in the file `NoiseFunction.py` will be discussed. The function that handles all implemented noise types is:

`Noise(noisemodel, T, dt, f, A)`

This function accepts the following values as input:

- noisemodel: string which represents the type of noise chose (e.g. sine, perlin, fractal);
- T: length of simulation time [s];
- dt: time interval between two passes [s];
- f: frequency of noise carrier [Hz];

- A: noise amplitude;

The output returns a vector of length equal to the number of samples (T/dt) containing for each element the noise value selected by "noisemodel" at that instant in time. This vector can be of the desired amplitude and frequency thanks to the parameters "f" and "A", and can then be added to the measurements of the various quantities to be noisemodelled.

First, the Noise function creates the time vector t of the appropriate length and initialises a vector of zeros that will then contain the noise produced. Next, the control on 'noisemodel' intervenes, allowing the right shape of the noise to be chosen. Below is a brief explanation of the implementation of Perlin Noise and Fractal Noise.

4.4.1 Perlin Noise implementation

The noise is created through the PerlinNoise function belonging to the perlin_noise library. An object of type PerlinNoise is created by a dedicated constructor, the argument of which allows the number of octaves to be entered. As far as octaves are concerned, their amount is calculated by multiplying the total time T by the frequency f. In fact, it has been assumed that the frequency of a PerlinNoise can be roughly represented by the number of octaves in the time unit.

Once this is done, the noise vector is calculated. The object of type PerlinNoise accepts real numbers as arguments, providing the desired noise values. A scan is then made of as many elements as necessary to fill the noise vector. If integers are supplied as arguments, the result returned by the PerlinNoise object is zero. To remedy this, values between 0 and 1 are provided by means of a scaling operation. The noise thus obtained is then adjusted to the desired amplitude A.

4.4.2 Fractal Noise implementation

Fractal Noise is obtained by adding 4 Perlin Noise of appropriate parameters. To do this, 4 objects of type PerlinNoise are created, with an increasing number of octaves (they are doubled each time). The amplitudes of the 4 noises are gradually reduced by halving them each time. The component with the largest amplitude is considered the carrier, and its number of octaves is calculated as in the previous section by multiplying T by f. One cycle allows these 4 components to be added together and generate the noise vector. At each iteration, a real number between 0 and 1 is given as the argument of the 4 noises, similar to

the Perlin Noise seen previously. It is useful to consider that each component has a multiple or submultiple number of octaves with respect to the others. For this reason and because of what was said in this chapter on Perlin Noise, even summing the 4 noises would still have zeros dividing the time axis. To overcome this problem, shifts are randomly generated which cause each of the 4 components to be out of phase with respect to the others. The result is that the zeros are no longer present systematically but randomly in the noise. Again, once the basic noise has been obtained, scaling is performed in order to obtain a signal of the desired maximum amplitude.

Chapter 5

Simulations

5.1 Introduction

This chapter will show four simulations in which the controller responds to inputs affected by both Perlin Noise and Fractal Noise, in order to provide an idea of a possible calibration of the constants Kp, Kd and Ki. As previously mentioned, no advanced control techniques for non-linear systems were used in order to optimise the controller, due to the great complexity of the system and since this was beyond the scope of the present discussion, which is limited to implementing a generic PID controller and a particular type of noise that will not necessarily be coupled in the future.

With regard to the various simulation parameters, the following list shows all the values used in each of the simulations.

- controller's parameters:
  - Kp = 0.07;
  - Ki = 0.000008;
  - Kd = 0.00000005;
  - output_type = 'Resistance';
  - setpoint = 0.7 mΩ
  - dt = 10 s;
  - maxSpeed = 75 cm/h;
  - zmin = 0.1 m;
  - zmax = 0.75 m.

- starting default parameters:
  - default_z1 = 0.3 m;
  - default_z2 = 0.325 m;
  - default_z3 = 0.29 m;
  - default_I1 = 116 kA;
  - default_I2 = 116 kA;
  - default_I3 = 116 kA;
  - default_CW_T = 25 cm;
  - default_sigma_arc = 383.33 S/m;
  - default_sigma_CW = 300 S/m;
  - default_sigma_SiC12 = 60 S/m;
  - default_sigma_SiC23 = 60 S/m;
  - default_sigma_SiC31 = 60 S/m;
  - default_sigma_coldCharge = 15 S/m;
  - default_sigma_hotCharge = 15 S/m;
  - default_sigma_lining = 1000 S/m.
- noise's amplitudes:
  - ERR_I1 = 8 kA;
  - ERR_I2 = 8 kA;
  - ERR_I3 = 8 kA;
  - ERR_CW_T = 15 cm;
  - ERR_SIGMA_CW = 100 S/m;
  - ERR_SIC12 = 40 S/m;
  - ERR_SIC23 = 40 S/m;
  - ERR_SIC31 = 40 S/m;
- noise carrier periods:
  - I1, I2, I3: 6 h;
  - CW_T: 1 h;

  - sigma_CW: 20 minutes;
  - sigma_SiC12, sigma_SiC23, sigma_SiC31: 6 h.

- total time of simulation: 24 h.

These parameters have not been varied from time to time in order to enable the simulations to be compared with each other. The simulations differ firstly by the type of noise used (Perlin or Fractal) and secondly by the number of input quantities to which this noise was applied. For both Perlin and Fractal, there is one simulation with only the three noisy electrode currents, and a second with noise on most of the system's inputs. A discussion of the results can be found in the relevant sections.

5.2 Simulation 1

This section will discuss the behaviour of the PID controller applied to the submerged arc furnace, with the currents I1, I2 and I3 of the three electrodes affected by Perlin-type noise. Below, some graphs will illustrate the noisy inputs and the results obtained.

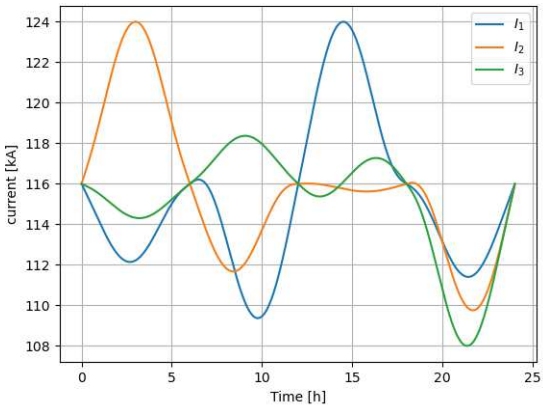


Figure 5.1: Noisy currents

The pattern of electrode positions on which the PID acts is shown in figure 5.2. The resistance value of the three controlled electrodes is shown in figure 5.3.

The result is quite satisfactory, response times are very fast and the controller never goes into saturation, maintaining a minimal error at steady state.

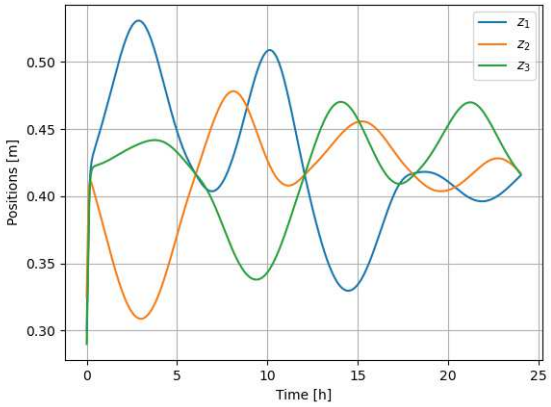


Figure 5.2: electrode's positions

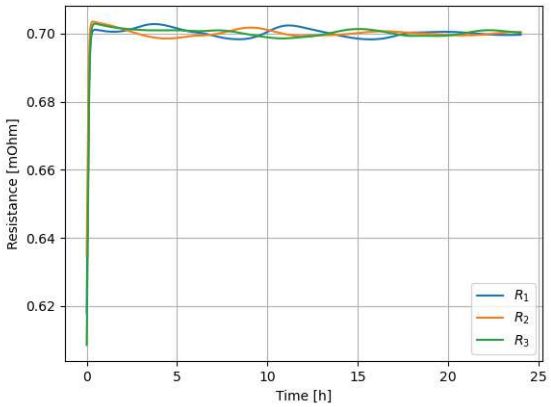


Figure 5.3: resistance

5.3 Simulation 2

In this case, in addition to the noise on the currents, sigma_CW, CW_T, and SiC12, SiC23, SiC31 became noisy. Below are graphs illustrating the noisy trends of the main inputs (in order not to burden them with too many pictures, it was decided not to attach the SiC trends). It is worth noting the rapidity of variation of sigma_CW, which has a carrier period of only 20 minutes, compared with the daily duration it is supposed to simulate.

This time the results are not as good as before. Several noisy inputs lead to unpredictable variations in the resistance parameter to be controlled, and in cases where effects due to different parameters overlap, it is very easy for the controller to go into saturation. It should be noted, however, that with the anti-windup control once the controller exits the saturation zone, it settles more easily than

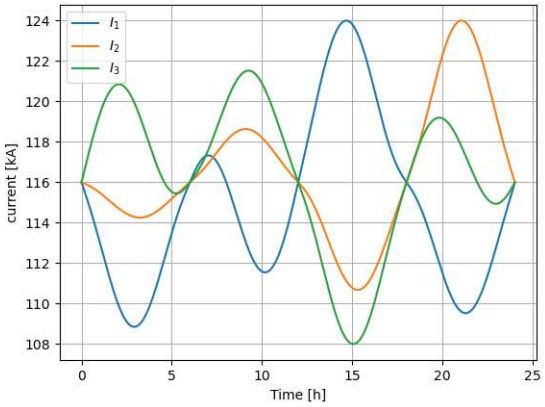


Figure 5.4: noisy currents

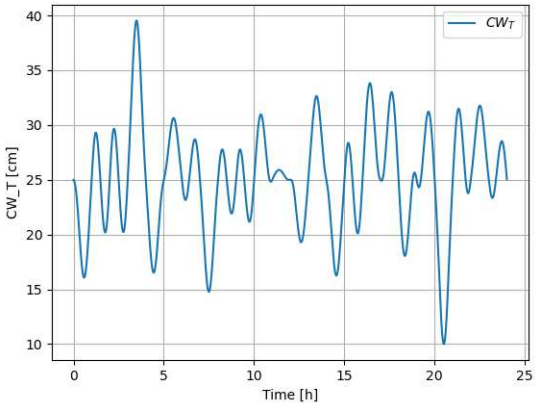


Figure 5.5: Crater Wall Thickness

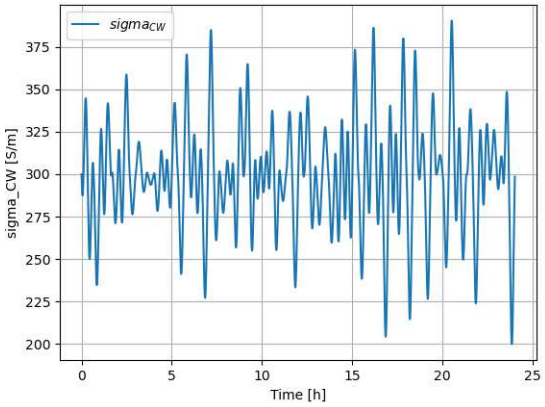


Figure 5.6: sigma CW

it would if this technique were not employed.

Below are graphs of the trend in electrode positions, on which the saturation

phenomenon can be observed, and the resistance trend, which in this case is not satisfactory. Further on, considerations will be made on possible solutions to remedy this problem.

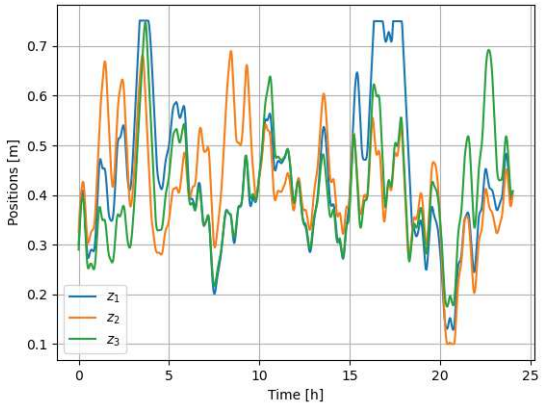


Figure 5.7: electrode's position

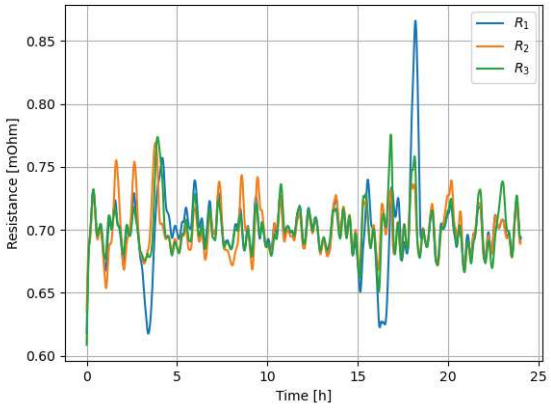


Figure 5.8: resistance

5.4 Simulation 3

Completely mirroring simulation 1, only the currents of the three electrodes were made noisy. In this case, however, the implemented Fractal Noise was applied. The currents are shown in figure 5.9.

Also similar to what was seen in the first section, the figures for the position of the electrodes and the resistance's value are shown below.

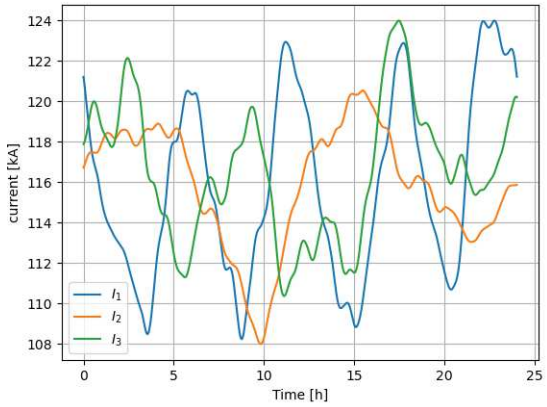


Figure 5.9: noisy currents

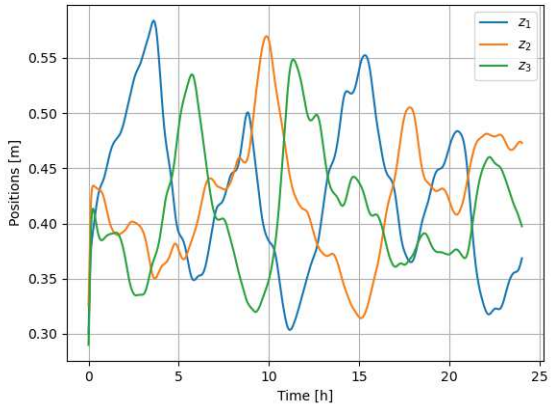


Figure 5.10: electrode's positions

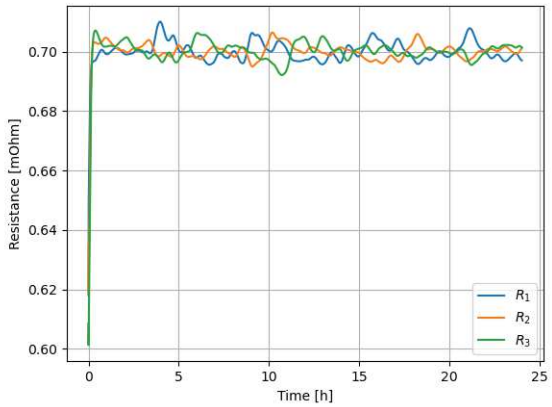


Figure 5.11: resistance

It is immediately noticeable that, although the results are acceptable, the controller struggles much more with Fractal Noise than with Perlin Noise. This

is due to the fact that the Fractal Noise is much more jagged, as a result the electrodes are subject to continuous changes in direction and position, often very fast. This effect already anticipates the fact that adding noisy parameters will lead to a very poor result when using Fractal Noise with this controller.

5.5 Simulation 4

The last simulation resembles the second one presented above, except that in this case Fractal Noise was used on the input variables (thus, in addition to the three currents, the three SiCs, sigma_CW and CW_T). As before, the input graphs are available from figure 1 to figure 4. Next, the two graphs depicting the electrode positions and the resistance trend can be found.

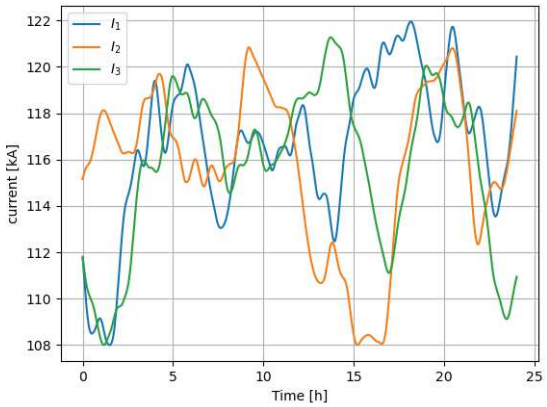


Figure 5.12: noisy currents

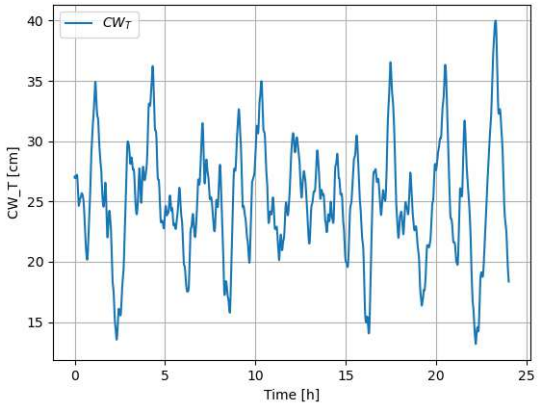


Figure 5.13: Crater Wall Thickness

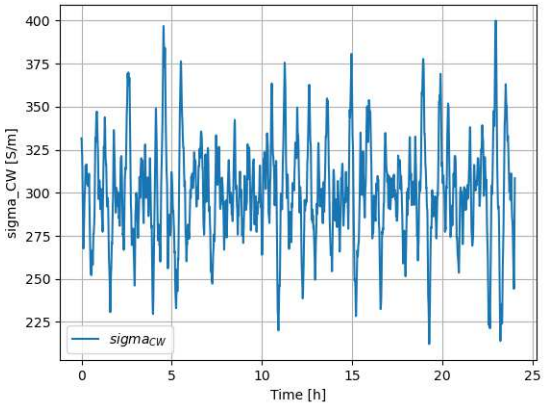


Figure 5.14: sigma CW

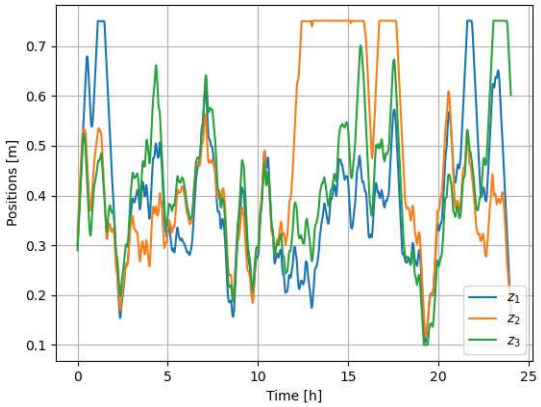


Figure 5.15: electrode's positions

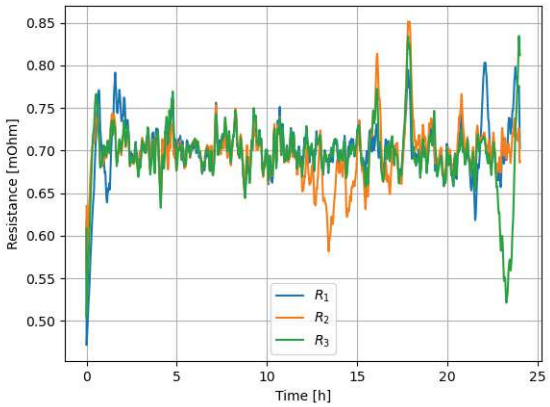


Figure 5.16: resistance

As can be seen, the result cannot be said to be satisfactory. There is a strong saturation especially in the position of electrode 2. This leads to intervals where

the electrode is at its highest but in practice its resistance is not properly controlled. This result is in line with what has already been mentioned in simulation 3. Fractal Noise is much more difficult to control than Perlin.

5.6 Conclusions

Some conclusions can be drawn from the previous four simulations. First, the results were satisfactory as long as the number of noisy parameters was reduced. Increasing the inputs subject to noise, as was easily expected, results in more difficult controllability. Furthermore, Fractal Noise proved to be much more difficult to control than Perlin. This too was easy to assume, since each Fractal is a sum of four different Perlin Noises. It should be considered in addition that one of the noisy parameters that make control more difficult is sigma_CW. Its period is very short, only 20 minutes, so when the noise is created, the associated octaves in both the Fractal and Perlin will be very small. This noise is therefore abrupt and difficult to follow optimally, also considering the fact that the electrodes must be subjected to a well-established speed limit. In the end, it is advisable to accompany the PID controller with some type of more advanced control, which allows it to be optimised to this type of noise. Furthermore, if the controller has access only to the electrode positions, it would be good to introduce another additional controller that intervenes when saturation occurs as it is done in the industrial plan with the tapping of the transformers. Such improvements can be made with a deeper understanding of the furnace meta model than is present in this Thesis. Please refer to a more detailed discussion for any future improvements.

Chapter 6

Code

Here is the code of the source file 'NoiseFunctions.py' containing all functions concerning the implemented controller and noise.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 25 09:44:14 2022

@author: mafr

Noise function modified on Tue Apr 26 21:23 2022 by Enrico Miotto

This script contains a Noise function, a controller function
and some functions used by the controller during its operation.

The noisemodel has three mode:
    sine: creates a sinewave vector
    perlin: creates a Perlin noise vector
    fractal: creates a fractal noise vector

The noise is currently added as an array with noisedata for the
   complete
time series to be simulated. This should be changed.


"""

import numpy as np
from perlin_noise import PerlinNoise
from random import random


```

```
28 def Noise(noisemodel,T,dt,f,A):
29     """
30     Noise function
31 
32     Parameters
33     ----------
34     noisemodel : type of noise (e.g. sine, perlin, fractal...);
35     T : length of simulation time [s];
36     dt : time interval between two passes [s];
37     f : frequency of noise's carrier component [Hz];
38     A : noise's amplitude;
39 
40     Returns
41     -------
42     noise : vector that contains the required noise
43             (it is full of zeros if noisemodel is not valid).
44 
45     """
46 
47     t = np.arange(0,T,dt)
48     noise = np.zeros(len(t))
49 
50     if noisemodel == 'sine':
51 
52         noise = A*np.sin(2*np.pi*f*t)
53 
54     elif noisemodel == 'perlin':
55 
56         perl = PerlinNoise(octaves=int(T*f))
57         y = [perl((i*dt)/T) for i in range(int(T/dt))]
58         scaleFactor = max(abs(max(y)), abs(min(y)))
59         noise = A*(np.asarray(y)/scaleFactor)
60 
61     elif noisemodel == 'fractal':
62 
63         noise = []
64         numberOctav = int(T*f)
65         noise1 = PerlinNoise(octaves=numberOctav)
66         noise2 = PerlinNoise(octaves=numberOctav*2)
67         noise3 = PerlinNoise(octaves=numberOctav*4)
68         noise4 = PerlinNoise(octaves=numberOctav*8)
69 
70         shift1 = 0.5*random()
71         shift2 = 0.5*random()
```

```
72          shift3 = 0.5*random()
73          shift4 = 0.5*random()
74
75          for i in range(int(T/dt)):
76              noise_val = noise1(shift1 + (i*dt)/T)
77              noise_val += 0.5*noise2(shift2 + (i*dt)/T)
78              noise_val += 0.25*noise3(shift3 + (i*dt)/T)
79              noise_val += 0.125*noise4(shift4 + (i*dt)/T)
80              noise.append(noise_val)
81
82          noise = np.asarray(noise)
83          scaleFactor = max(abs(max(noise)), abs(min(noise)))
84          noise = A*(noise/scaleFactor)
85
86      elif noisemodel == 'file':
87          temp = np.fromfile("FractalNoise.dat", dtype=float)
88
89          noise = temp[0:T:dt]
90
91      return noise
92
93
94
95  def controller_function(output_type,
96                          setpoint,
97                          last_value_out,
98                          z,
99                          dt,
100                         integral,
101                         lastError,
102                         maxSpeed,
103                         zmin,
104                         zmax,
105                         Kp = 0.07,
106                         Ki = 0.000008,
107                         Kd = 0.00000005):
108     """
109     Created on Tuesday April 19 20:09 2022
110
111     @author Enrico Miotto
112
113     Implementation of a PID controller which acts on electrode's
        position.
114
```

```
115     Parameters
116     ----------
117     output_type : the type of output needed
118                   (e.g. power, resistance, ecc...);
119     setpoint : setpoint that the PID will try to achieve;
120     last_value_out : last measured output value;
121     z : last position of the electrode [m];
122     dt : time interval between two passes [s];
123     integral : integral of the past error;
124     lastError : last value of the error;
125     maxSpeed : maximun speed that the electrode can reach [cm/
    hour];
126     zmin : minimum height [m];
127     zmax : maximum height [m];
128     Kp : The value for the proportional gain Kp;
129     Ki : The value for the integral gain Ki;
130     Kd : The value for the derivative gain Kd.
131
132     Returns
133     -------
134      z + dh : updated position of the electrode;
135      integral : updated value of error's integral;
136      lastError : updated value of the error for the next
    iteration.
137
138     """
139
140     error = setpoint - last_value_out
141     dh_max = (maxSpeed * dt) / 360000 # [m]
142
143     # INTEGRAL
144     # updates the integral of the error which will be returned
145     integral += Ki * error * dt
146     integral = clamp(integral, -dh_max, dh_max) #avoid integal
    windup
147
148     # DERIVATIVE
149     derror = error - lastError
150
151     # dh is the quantity to add to the electrode's height
152     dh = error * Kp + integral - (derror / dt) * Kd # [m]
153
154     # checks if the controller has to act on the position
155     if not isPositionValid(z, dh, zmin, zmax):
```

```
156         dh = 0
157
158     # checks if the max speed is reached
159     dh = clamp(dh, -dh_max, dh_max)
160
161     # updates the error which will be returned
162     lastError = error
163
164     return z + dh, integral, lastError
165
166
167 def isPositionValid(z, dh, zmin, zmax):
168     """
169
170     Created on Thursday April 28 15:50 2022
171
172     @author Enrico Miotto
173
174     function that checks if the position of one electrode is
        valid.
175
176     Parameters
177     ----------
178     z : actual position of the electrode;
179     dh : further increment/decrement of the position;
180     zmin : minimum height;
181     zmax : maximum height.
182
183     Returns
184     -------
185     true if the position is valid, false otherwise.
186
187     """
188     return (z >= zmin and z <= zmax) or (z < zmin and dh > 0) or
        (z > zmax and dh < 0)
189
190 def clamp(value, lower, upper):
191     """
192     Created on Mon May 23 11:59 2022
193
194     @author Enrico Miotto
195
196     function that checks if a quantity has exceeded
197     its upper or lower bound.
```

```
198
199     Parameters
200     ----------
201     value : TYPE
202         DESCRIPTION.
203     lower : lower bound.
204     upper : upper bound.
205
206     Returns
207     -------
208     value if it's between the bounds,
209     higher or upper bound otherwise.
210
211     """
212     if value is None:
213         return None
214     elif (upper is not None) and (value > upper):
215         return upper
216     elif (lower is not None) and (value < lower):
217         return lower
218     return value
```

Listing 6.1: Didascalia.

References

[1] Åström K. J., *Control System Design*, 2002.

[2] Cecchin F., *Regolatori PID autotunig per il controllo della temperatura*, 2014.

[3] Monsen I., *Bachelor Thesis*, 2022.

[4] Perlin K., *An Image Synthesizer*, 1985.

[5] Perlin K., *In the beginning: The Pixel Stream Editor*, Retrieved June 30, 2022.

[6] *perlin-noise*, `https://pypi.org/project/perlin-noise/`, version 1.12.

[7] *SAFECI - Silicon Furnace Electrical Conditions Metamodel*, `https://safeci.web.norce.cloud/`, v.2022.05.25.

[8] Sparta M., Fromreide M., Risinggård v. K., Halvorsen S. A., *Electrical conditions in submerged arc furnaces: an online simulator*, 2022.

[9] Testolin G., *Controllori PID e tecniche "anti wind-up"*, 2013.

[10] *Value Noise and Procedural Patterns: Part 1*,

`https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples`,

last consultation 30/06/2022.