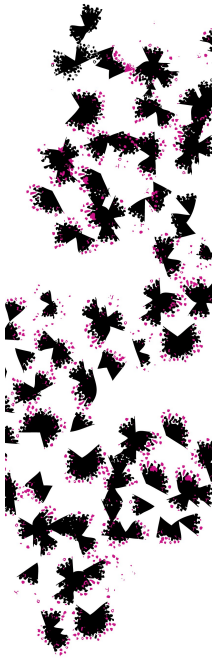


Learning Feed-Forward Control with the Python Scikit-Learn Library

E.A. (Elise-Ann) Schrijvers

MSc Report

Committee:
Dr.ir. J.F. Broenink
Dr.ir. T.J.A. de Vries
Dr.ir. G.M. Bonnema

October 2017

045RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

The research of this thesis is about using a learning feed-forward controlled system in a platform independent way. To achieve this, the feed-forward part of the control system is implemented in Python while the general control system is within the 20-sim simulation environment. The implementation of LFFC in Python is relatively simple due to the existence of the Scikit-learn library. This library enables the use of a B-spline network (function approximator).

Communication between both environments is achieved by setting up a network connection. To that end, data will be serialized and packed by the Protocol Buffer library from Google and ZeroMQ. The data can now be sent over the network in a proper and structured way.

The 1-dimensional time-indexed LFFC is implemented twice. One is completely built-up in the environment of 20-sim and the other has its feed-forward part built-up in Python. A 1-dimensional state-indexed LFFC in Python is considered as well. All implementations are demonstrated by assuming an ideal linear motor model (moving mass) representing the plant of the control system.

In the case of the two dimensional state-indexed LFFC a plant model is used that includes two different phenomena. One phenomena was considered in the 1-dimensional case as well, i.e. the inertia of the mass. The second phenomena is non-ideal and depends on the position of the linear motor, described as the cogging. This type of LFFC is implemented in two different ways, i.e. 2x a 1-dimensional BSN (parsimonious LFFC) and 1x a 2-dimensional BSN. Both imply the use of a different trainings method. The first approach trains one BSN at a time and in such a way that only one plant influence is dominant and will be learned. The second approach will try to learn two plant influence at the same time by using only one BSN. A demonstration is given of the first approach by assuming a plant model that incorporates inertia and position dependent cogging (non-ideal linear motor model). The second approach is not demonstrated and further research is required (even though this implementation is not preferred as it has to deal with the curse of dimensionality).

Contents

Nomenclature

1D 1-dimensional

2D 2-Dimensional

ANOVA ANalysis Of VAriance

BSN B-Spline Network

CA Constant Acceleration

CV Constant Velocity

CY Constant Jerk

DLL Dynamic Link Library

FA Function Approximation

LC Learning Controller

LFFC Learning-Feed Forward Control

LMMS Linear Motor Motion System

NM No Movement

TP Transition Point

1 Introduction

1.1 Context

Learning Feed-Forward Control (LFFC) has proven to be a powerful control architecture that has much potential for control of mechatronic systems, Velthuis (2000). A LFFC system consists of a model-based feedback component and a feed-forward component that has learning abilities, i.e. it consists of a function approximator. The feedback part is a typical PD-type controller.

Starrenburg et al. (1996) started studying LFFC and later also Velthuis (2000) using B-spline Networks (BSN) on repetitive motions. He performed a stability analysis on the LFFC and came up with rules to be able to properly select design parameters for such type of motions. Besides the LFFC for repetitive motions he also study non-repetitive motions. In this part he addressed multi-dimensional B-spline networks and introduced parsimonious LFFC. The latter was a solution to overcome the problems that go along with the curse of dimensionality. He used among others a linear motor motion system (LMMS) as an illustrative example for his study. The study from Velthuis (2000) is used as the base for this thesis.

1.2 Problem Statement

Simulations are commonly done in a Windows environment, for which real-time aspects are not so important, while experiments are done in a real-time Linux environment. This assignment addresses to find a solution to interact between the feed-forward part of a control system and a general PD-controlled system in a platform independent way. Therefor a network connection has to be incorporated into the control system.

The problem encounters to create and application that:

- is as straight forward as possible while at the same time the LFFC features (high performance and high robustness) are still maintained
- overcomes the following drawbacks that appear in currently available applications:
    1. the computational intensiveness of the learning process
    2. impossibility to combine LFFC with real-time control
    3. restricted use in the type of function approximator that can be used (BSN). A BSN might not always result in the optimal combination with LFFC
    4. testing the LFFC and painlessly transfer to a realization environment is not that simple

The envisioned solution is:

- to use Python Scikit-learn for the learning process (function approximation (FA)), as it provides BSN and other FA's and it is platform independent. Unfortunately, this library is not directly suitable for real-time applications.
- to connect to Python via a network, such that this can be done both in simulation and in practice. But also to do learning at another computer than real-time control.
- to demonstrate LFFC in the simulation environment of 20-sim, followed by separating the feed-forward controlled part from the simulation environment and implement it in Python.
- to demonstrate the LFFC using 1- and 2-dimensional LFFC. To illustrate this a linear motor model is controlled

1.3 Thesis Outline

In Chapter 2 some background is provided about learning feed-forward control and how function approximation is performed using B-splines. An introduction is given about implementing B-spline networks in the environments of 20-sim and Python. The background chapter is concluded with describing the illustrative example used for the thesis: a linear motor motion system.

In Chapter 3 the network communication set-up between 20-sim and Python is explained. Two protocols are discussed that perform data packaging and the data transfer between both ends of the communication network (Protocol Buffer from Google and ZeroMQ). The chapter is concluded with tests that will verify if both protocols work individually, but also if both protocols perform correctly when both are combined.

Chapter 4 demonstrates the use of 1-dimensional LFFCs. First a time-indexed LFFC is discussed implemented in the environments of 20-sim and Python. The second part of the chapter demonstrates the use of 1-dimensional state-indexed LFFC in Python.

A 2-dimensional LFFC is discussed in Chapter 5. This chapter distinguishes between the use of two 1-dimensional B-spline networks (parsimonious LFFC) and one 2-dimensional BSN. The latter is only briefly discussed and is not demonstrated with simulations. More research is required about this topic.

A conclusion and abbreviations are presented in Chapter 6.

2 Theoretical Background

This chapter provides the reader from information in order to understand the subjects treated in the thesis. The information starts with explaining what learning feed-forward control (LFFC) is and why it is used. Depending on the inputs of the feed-forward part a time-indexed or state-indexed LFFC is preferred, both types are treated.

An important part of LFFC is the function approximator, although there are many possible types of function approximators only the B-spline network (BSN), de Kruif and de Vries (2000) will be treated as this is the method used in the thesis.

Two different BSN implementations will be discussed, one describes the implementation using the built-in B-spline editor from the simulation software 20-sim and the other describes the implementation using the Scikit-learn library from Python.

In the last section an illustrative example is given. The example used is a plant that represents a model of a linear motor motion system. The plant is controlled by a PD-type feedback controller that is tuned to meet certain specifications. The LMMS is a nice example because it is of an actuator type that is used increasingly in mechatronic systems and at the same time it suffers from cogging, de Kruif and de Vries (2000), a non-linear disturbance that lends itself well for LFFC and is not easily compensated in feedback. The model presented is used as the basis for the models used later on in the thesis.

2.1 Learning Feed-Forward Control

In the development of high-tech products (among others electro-mechanical motion system) the product performance is of great importance and superiority is expected. The performance of such a system is influenced by both the mechanical design and the tuning of the controller. The moment the systems performance must be improved most commonly it is chosen to change the controller in stead of making structural adaptations. Controller changes are more easily to implement as in most situations software adjustments are sufficient.

The design of a controller is based on a plant model and its performance depends on the accurateness of the model used. The more accurate the plant model the better the performance of the controller. The following problems might be encountered when modeling a plant, Harris et al. (1993):

- → The system is too complex to understand or to represent in a simple way
- → Model evaluation is difficult (often due to non-linear effects) or to expensive
- → The plant is subjected to large environmental disturbances, which makes it hard to predict
- → The plant parameters might be time-varying

In situations the model is not available or parameter predictions are not possible, learning control can be applied. From Velthuis (2000) a definition of a learning controller is presented:

"A learning controller is a control system that comprises a function approximator of which the input-output mapping is adapted during control, in such way that a desired behaviour of the controlled system is obtained."

Learning feed-forward controllers can be divided into two categories, i.e. the time-indexed and the state-indexed LFFCs. A time-indexed LFFC is characterized by having one input only and is applied for repetitive tasks, i.e., repeated motions having a fixed path and a fixed period. The input supplied to the BSN is the periodic motion time T_P and the B-splines are divided along the input range from $[0, T_P]$. Since the learning controller uses only one input there is no need

to concern about the curse of dimensionality, O'Flaherty and Egerstedt (2015). The structure of a time-indexed LFFC is shown in Figure 2.1.

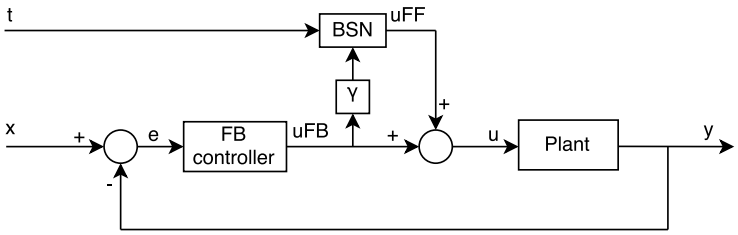


Figure 2.1: Structure of a 1-dimensional time-indexed LFFC

The main drawback of a time-indexed LFFC is that it is for repetitive motions only. This means that the learning controller is useful for one motion pattern only. If a different motion is wished to be performed the learning controller needs to start its learning process all over again and it has lost its ability to track the old motion.

To overcome this drawback a state-indexed LFFC can be used. This type of LFFC is supplied with one or more reference signal(s), i.e. position x, velocity v and/or acceleration a, Velthuis et al. (1998). As a result, the learning controller can now be applied for both repetitive and non-repetitive motions. In Figure 2.2 the structure of a 1-dimensional state-indexed LFFC is given (acceleration as input).

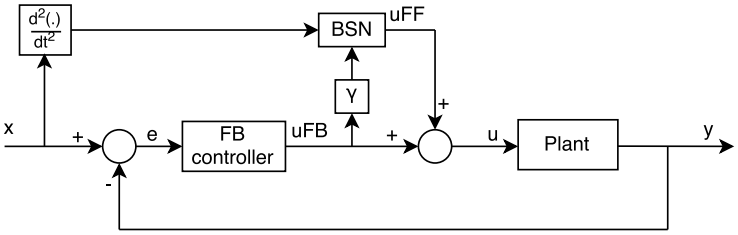


Figure 2.2: Structure of a 1-dimensional state-indexed LFFC (BSN input: a)

A drawback of this type of LFFC is that for large number of BSN inputs the curse of dimensionality starts to play a role. In order to minimize this problem parsimonious modeling techniques can be used, according to Bossley and Harris (1997) which stated:

"The best models are obtained using the simplest possible, acceptable structure that contain the smallest number of parameters".

Several strategies exists to obtain parsimony, Velthuis et al. (1998):

- **Minimize the number of B-splines on each input domain**
  By selecting the number of B-splines as low as possible, the number of network weights will be minimized, the smallest possible training set can be used and the generalizing ability is as good as possible.

  The generalizing ability defines how well the learning controller performs when trajectories are supplied that are "close to each other". A poor generalizing ability is defined the moment several "close to each other" trajectories are supplied, but very different network output signals are obtained.

- **Split high-dimensional BSN up into lower-dimensional BSNs**
  By reducing the dimension of a B-spline network the number of required B-splines will

drop exponentially. (The number of B-splines and the network dimension are exponentially related.)

A high-dimensional BSN can be splitted up by writing the target function in the ANalysis Of VAriance (ANOVA) representation. Given a n-dimensional function $f(.)$:

$$y = f(x_1, x_2, ..., x_n) = f_0 + \sum_i f_i(x_i) + \sum_{i,j} f_{i,j}(x_i, x_j) + ... + f_{1,2,...,n}(x_1, x_2, ..., x_n) \tag{2.1}$$

in which $f_i(.), f_{i,j}(.), ...$ the univariate, bivariate, ... additive components of $f(.)$. As an example the following function is assumed:

$$y = f(x_1, x_2, x_3) = f_1(x_1) + f_{1,3}(x_1, x_3) + f_{2,3}(x_2, x_3) \tag{2.2}$$

Figure 2.3 is used to demonstrate Equation 2.2. To the left the structure of a 3-dimensional BSN is shown and to the right the structure is shown having similar abilities but lower dimension. The latter uses one 1-dimensional BSN and two 2-dimensional BSNs.

The 3-dimensional BSN requires $N_{tot} = N_1 N_2 N_3$ network weights and the lower dimensional structure requires $N_{tot} = N_1 + N_1 N_3 + N_2 N_3$, in which N_i the number of 1-dimensional B-spline functions on domain i. The larger N_i the more beneficial it is to split up a multidimensional BSN structure.

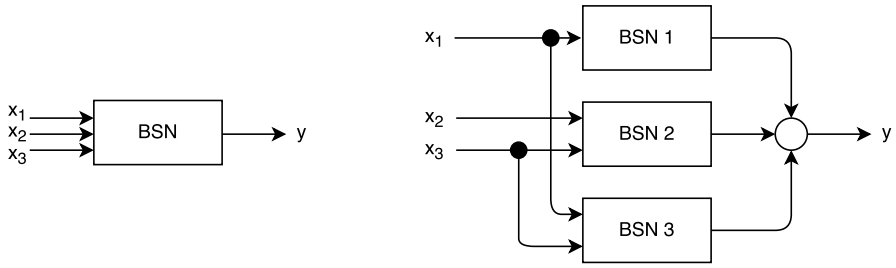


Figure 2.3: Two equal BSN structures using, 1 BSN (left) and 3 BSNs (right)

Depending on the structure of the LFFC special attention might be required for the way the learning controller will be trained. Structures that only have a single 1-dimensional structure do not need special attention. The moment the structure has to learn more than one feature of the plant, proper learning becomes more challenging. In order to train a parsimonious LFFC, Buijssen (2001) proposed to train one BSN at a time (proposition 4.1). The reference motion used for the training must be chosen in such a way that the desired output of one of the untrained BSNs is temporarily dominant. This way only the weights of the dominant BSN are adapted during the training and the others remain constant. To achieve this, the following step-by-step plan can be used:

1. Use a trainings motion for which one target signal of an untrained BSN is dominant
2. Train selected BSN until convergence and use other trained BSN as control signal
3. Back to step 1 if untrained BSNs exist, otherwise the training is finished

2.2 Function Approximation with B-splines

This section is started off with representing the definition of a function approximator (by Velthuis (2000)):

"A function approximator is an input-output mapping determined by a selected function $F(., \boldsymbol{w})$, of which the parameter vector $\boldsymbol{w}$ is chosen such that a function $f(.)$ is "best" approximated."

The learning controller is implemented with a function approximator. A wide variety of function approximators exists, like neural networks, neuro-fuzzy networks and look-up tables, Polycarpou and Ioannou (1992). For this thesis a B-spline Network is used because the current application of LFFC (in 20-sim) and the application to be newly built (in Python) both have a B-spline network available. This way, both implementations can be compared on performance and easiness of use (and design). The approximation is performed by forming B-spline curves and is used in the modus "indirect learning control". This means that the function approximator learns the model of the plant under control by adaptation of the approximator in order to minimize the cost function of the prediction error. A B-spline network has advantages (✓) and disadvantage (×):

- ✓ **No local minima**
  The BSN output is a linear function of the weights and the initial weights used do not influence the final tracking accuracy.

- ✓ **Local learning**
  The in- and output mapping of the BSN can be adapted locally as the support of a B-spline can be compact. During a training only a small number of weights contribute to the output. This is caused by the fact that only the weights of those B-splines are adapted. This is beneficial for the rate of convergence of the BSN.

- ✓ **Tunable precision**
  The B-spline distribution determines the smoothness of the in- and output mapping. To achieve a smoother approximation (for instance if the target signal contains more high-frequency data) either the support of the B-spline can be chosen larger or the degree of the B-splines can be increased.

- × **Large number of network weights**
  A highly non-linear function has to be mapped by a B-spline network if the plant has dynamics that are described by highly non-linear components. To be able to map those non-linearities accurately a lot of computer memory is required together with large computational cost, which is especially not desired in real-time control. (curse of dimensionality)

- × **Large training set**
  The network weights that are indexed by the networks input will only be adapted for a specific reference motion. This means that the moment a large number of network weights must be adapted a large number of trainings motions must be supplied to the network. As a result, the total training time of the network will increase. (curse of dimensionality)

- × **Poor generalizing ability**
  In order to accurately approximate non-linear plant behavior it might be required to select narrow B-splines. Though, in combination with trajectories that are "close to each other" the narrow B-splines may result in different network output signals. Therefor large training sets has to be supplied to the approximator in order to notice beneficial effects. (curse of dimensionality)

In order to set-up a function approximator (BSN), the following design choices have to be made:

1. **Inputs of the BSN**
   The curse of dimensionality is a factor that is related to the number of inputs of the BSN. For high dimensions large number of BSN network weights are incorporated, large train-

ings sets are required and its generalizing ability will be pore. And therefor high system dimensionality should be avoided.

Depending on the motion to be performed two types of inputs can be chosen. For repetitive motions the periodic motion time is commonly used and for non-repetitive motions the reference position x and/or derivatives thereof ($\dot{x} = v$ and $\ddot{x} = a$) can be supplied to the BSN.

2. **B-spline distribution of the BSN('s)**
The output of the BSN is the weighted sum of the B-spline evaluations, as a result the accuracy of the approximation depends on the number of B-splines and their locations. The target signal has to be approximated and based on this signal a low number of "wide" splines or a large number of "small" B-splines can used. The latter is required for strongly fluctuating signals. See Figure 2.4 for a target signal, a B-spline distribution and the corresponding BSN approximation.

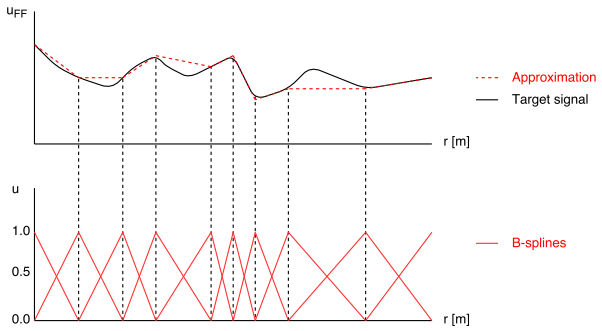


Figure 2.4: Target signal and the approximated signal by a B-spline network

In Figure 2.5 two target signals ares shown, one containing high frequencies and one low frequencies. For both target signals a B-spline approximation is shown using equal number of B-splines and in both situations uniformly distributed splines.

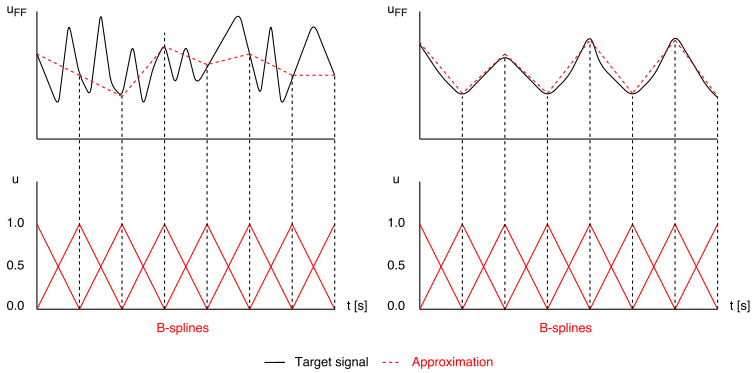


Figure 2.5: B-spline approximation, left) high frequency target signal and right) low frequency target signal

By increasing the number of basis functions (decreasing the B-spline width) the learning controller is able to approximate high frequency elements as well. In selecting a too small B-spline width it might be possible that 1) noise and unwanted high frequency signals will also be approximated and 2) the approximation diverges and the system becomes unstable, Bishop H. Robert H. Bishop (2007).

3. **Selection of learning mechanism**
   The learning mechanism of the approximator specifies the adaptation of the network weights. Adaptation can take place after each sample ("on-line learning") or after completion of a motion ("off-line learning"). Both methods have their own learning rule:

   Learn after a sample:

$$\Delta w_i = \gamma u_i(\mathbf{r}) e(\mathbf{r}) \tag{2.3}$$

   Learn after completed motion:

$$\Delta w_i = \gamma \cdot \frac{\sum_{j=1}^{N_s} u_i(\mathbf{r}_j) e(\mathbf{r}_j)}{\sum_{i=1}^{N_s} u_i(\mathbf{r}_j)} \tag{2.4}$$

   with, $\mathbf{r}_j$ BSN input
   $u_i(\mathbf{r}_j)$ membership of i-th B-spline, for which $u_i(\mathbf{r}_j) \in [0,1]$
   Δw_i adaptation of the weight of the i-th B-spline
   γ learning rate, for which holds $0 < \gamma \leq 1$
   $e(\mathbf{r}_j)$ network approximation error, the output of the feedback controller u_{FB}
   N_s number of input samples

4. **Selection of the learning rate**
   After a complete motion is performed, the learned data is applied to the system the moment the next motion starts. The learning rate of the approximator is related to the number of motions that needs to be performed in order to let the learning mechanism converge. A large value makes the convergence fast but may also increase the systems sensitivity to noise and/or cause instability.

Although the purpose of the research is not to implement a control system with a learning feed-forward controller with optimal performance it is important to have a look at the stability of the feed-forward part.

The feed-forward controller is said to be stable if an arbitrarily chosen initial feed-forward signal will not cause an unbounded output of the plant. The initial feed-forward signal is determined by the initial values of the weights within the B-spline network. For a stable feedback system the only way to observe an unbounded output is the moment the feed-forward signal u_{FF} becomes unbounded. This implies that at least on weight has become infinitely large. In order to achieve a stable system the weights must be adapted with care such that their values remain bounded.

Later on in the thesis, simulation experiments are described. Those simulations have BSN network settings (number of B-splines and the learning rate) that are selected in such a way that at first sight no instable behavior seems to occur. Though, it might be possible that by extending the duration of a simulation or by increasing the number of runs in a multiple run simulation experiment that instability will occur. Designing a perfect LFFC using BSN is beyond the scope of this thesis.

2.2.1 B-spline Basis Functions

The domain of a B-spline curve is subdivided into knots and the $m+1$ knots together form the knotvector U, for which holds that $u_0 \leq u_1 \leq ... \leq u_m$. Each knot divides the interval $[u_0, u_m]$ into half-open knot spans, for instance the i-th knot span on the half-open interval $[u_i, u_{i+1})$. Simple knots are knots appearing only once and knots that appear k times are knots having a multiplicity of k. The spreading of the knots over the B-spline domain can either be uniform

(equally distributed) or non-uniform (not equally distributed). The B-spline distribution used in the simulations all have simple knots $k = 1$, besides the boundary knots. The knots at the boundary have multiplicity of $p+1$, in which p is the degree of the splines.

Each B-spline basis function is defined within the domain $[u_0, u_m]$. The basis functions are used as weights and shapes the approximation of the curve. The basis functions are described by the so called Cox-de Boor recursion formula:

$$N_{i,0} = \begin{cases} 1 & if\, u_i \le u \le u_{i+1} \\ 0 & otherwise \end{cases} \tag{2.5}$$

$$N_{i,p} = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \tag{2.6}$$

with, p the degree of the basis functions
$N_{i,p}(u)$ the i-th B-spline basis function of degree p
u_i the i-th knot

The shape of the basis functions is defined by its degree and is used to set the maximum achievable smoothness for the curve approximation. First order (zero degree) basis functions are described by Equation 2.5 and have only one constant parameter. This function can be seen as a step function, $N_{i,0}(u)$, which means that it has exactly one non-zero interval and a discontinuity at u_{i+1}, see Figure 2.6.

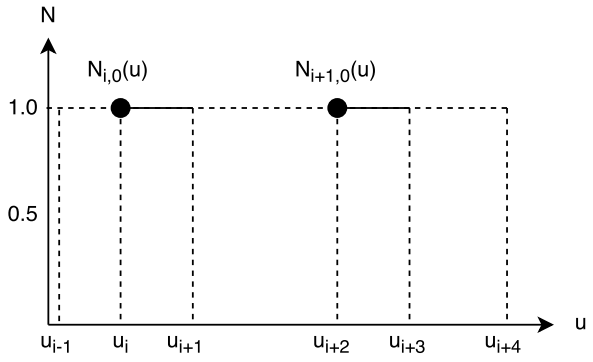


Figure 2.6: Non-zero parts of B-spline basis functions of degree 0 (order 1)

First degree basis functions can be described by two linear segments (triangular shape, see Figure 2.7) and its corresponding equations can be derived from Equations 2.5 and 2.6:

$$N_{i,1} = \frac{u - u_i}{u_{i+1} - u_i} N_{i,0}(u) + \frac{u_{i+2} - u_i}{u_{i+2} - u_{i+1}} N_{i+1,0}(u) \tag{2.7}$$

$$N_{i,1} = \begin{cases} \frac{u-u_i}{u_{i+1}-u_i} & u \in [u_i, u_{i+1}) \\ \frac{u_{i+2}-u}{u_{i+2}-u_{i+1}} & u \in [u_{i+1}, u_{i+2}) \\ 0 & elsewhere \end{cases} \tag{2.8}$$

The basis functions of degree one have two non-zero parts, defined on the intervals $[u_i, u_{i+1})$ and $[u_{i+1}, u_{i+2})$, see Figure 2.7. Both intervals together are referred to as the support of basis function $N_{i,1}(u)$. The functions are linear on both parts of the support interval and the location and slope are fully determined by the distribution of the u_i's.

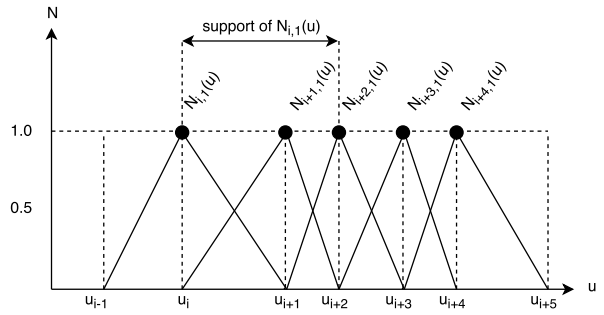


Figure 2.7: Non-zero parts of B-spline functions of degree 1 (order 2)

Second degree basis functions (and higher) are superpositions of multiple quadratic basis functions. The larger the degree the more smooth the function approximation can be. In Figure 2.8 an overview is given of the basis functions of degree 0, 1, 2 and 3 (equal to order 1, 2, 3 and 4).

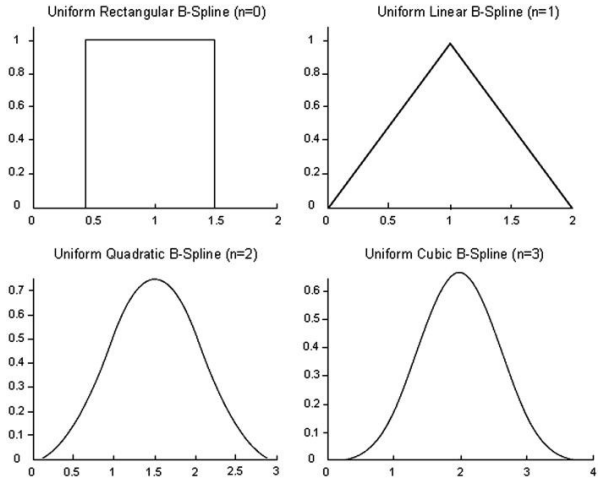


Figure 2.8: B-spline basis functions of degree $n = 0, 1, 2$ and 3

In order to determine a basis function that has a degree larger or equal than 1 the triangular computation scheme can be used, see Figure 2.9. In the scheme knot spans are listed in the first column seen from the left and basis functions with increasing degree (started from zero) are shown in the columns to the right of the knot spans.

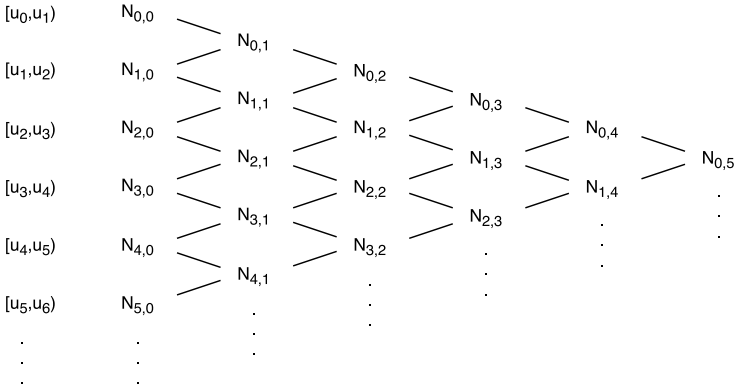


Figure 2.9: Triangular computation scheme

To make use of the triangular scheme more clear, assume that the non-zero domain of basis function $N_{1,3}(u)$ has to be determined. By trace back the scheme in the direction towards the first column all required basis functions will be calculated, see Figure 2.10.

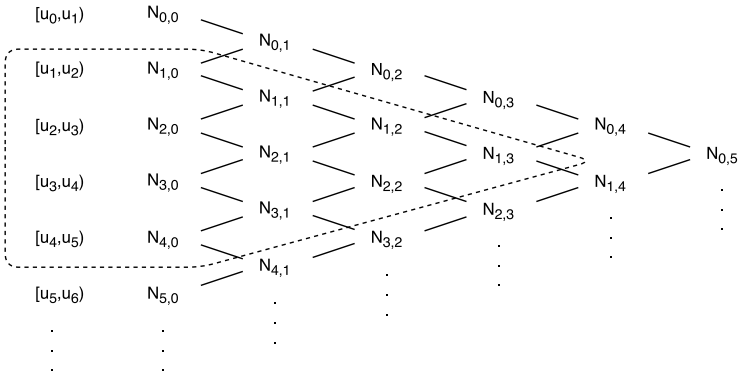


Figure 2.10: Triangular computation scheme, trace back for basis function $N_{1,3}(u)$

An elaborate explanation of the B-splines basis functions can be found in Appendix B.1.

2.2.2 Computing Coefficients

For a given clamped B-spline curve of degree p the re-currency relation of Equation 2.8 can be used. Though, for large degree this method can be time consuming and inefficient. As it might occur that in a series calculation some coefficients are calculated multiple times.

Assume that u is within the half open knot span $[u_k, u_{k+1})$ then at most $p+1$ basis functions of degree p are non-zero $\big(N_{k-p}(u), N_{k-p+1}(u), N_{k-p+2,p}(u), ..., N_{k-p,p}(u)\, N_{k,p}(u)\big)$ and the only non-zero basis function is $N_{k,0}(u)$. By having this basis function as the starting point for the triangular computation scheme and from thereon work along the columns until all the required $p+1$ coefficients are known, see Figure 2.11

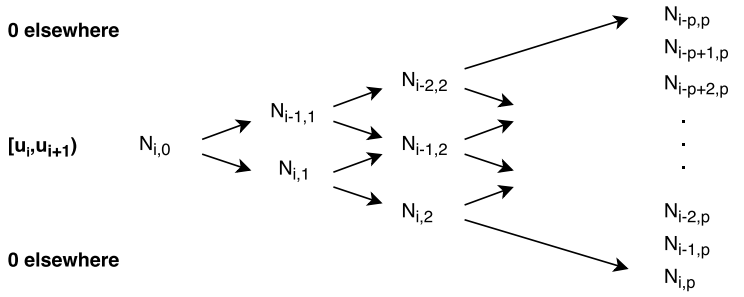


Figure 2.11: Triangular scheme of non-zero B-spline coefficients

2.3 B-spline Network Tools

For the thesis the 20-sim (using the built-in B-spline editor) and Python (using the Scikit-learn library) implementations of the B-spline networks are compared. In this section some information is provided about both.

2.3.1 B-spline Network with 20-sim B-spline Editor

The software 20-sim is implemented with a built-in B-spline network editor, 20simBSN (2017), see Figure 2.14. The B-spline network relates k inputs to a single output y on a certain domain of the input space. The structure of the network can be seen to consists of four layers, i.e. one input layer, two hidden layers and one output layer.

Both hidden layers consists of n nodes of which each node has only one input. Fed to the nodes of the first hidden layer are N-th order basis functions F and to the nodes of the second hidden layer a function G. Function G multiplies the input by a certain weight. The output node gives the resulting sum of all the node outputs of the second layer.

To make this more clear, a one-dimensional B-spline network is assumed having a single input. The structure of this network is shown in Figure 2.12.

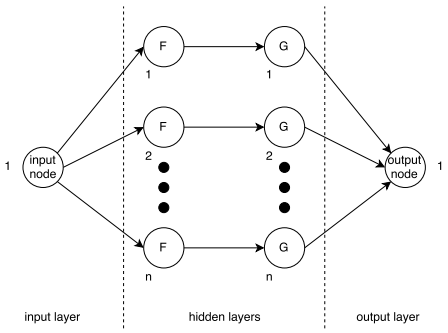


Figure 2.12: B-spline network structure of 20-sim

For a properly spaced spline domain it is possible to approximate every one dimensional function, see Figure 2.13.

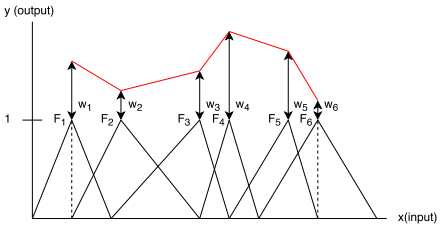


Figure 2.13: Function approximation with B-splines in 20-sim

Training of the network is done by comparing the network output y with the desired output y_d. The observed error between both is used to adapt the weights and the rate at which the adaptation takes place is defined by the learning rate γ. A quick adaptation can be achieved by using a high learning rate, though for an increased risk of unstable behavior. For $\gamma = 0$ the learning is disabled and weight adaptation will not take place.

Besides the learning rate the parameters to be set in the B-spline editor (see Figure 2.14) are the order of the B-splines, the number of splines and the lower and upper input data value. Within the editor it is possible to select the networks learning mode (learning at each sample or learning after leaving a spline) and the network type (continuous time or discrete time).

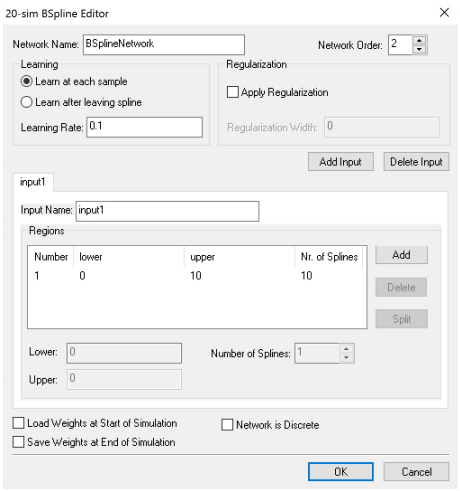


Figure 2.14: The B-spline editor window of 20-sim

The mode "learning at each sample" updates the network weights after each sample (according to Equation 2.9a). For a certain input x only a few splines have $F_i(x) \neq 0$, which means that at each sample only a few weights will be adapted. The mode "learning after leaving a spline" keeps track of input x and its corresponding non-zero splines $F_i(x)$. Samples of non-zero splines are stored and only after the input has left the region of a non-zero spline its weight will be adapted according to Equation 2.9b.

$$\Delta w_j = \gamma \cdot (y_d - y) F_j(x) \tag{2.9a}$$

$$\Delta w_j = \gamma \cdot \frac{\sum_{i=1}^{n} \left(y_{d,i} - y_i\right) \cdot F_j(x_i)}{\sum_{i=1}^{n} F_j(x_i)} \tag{2.9b}$$

In which Δw_j represents the adaptation of weight w_j, γ the learning rate, $F_j(x)$ the basis function of sample j, x the input, y_d the desired output and y the network output.

The calculated weights can be saved to file after a simulation experiment has been finished and weights can be loaded from file before the start of a simulation experiment. This makes it possible to use each run different initial data in a multiple run simulation experiment.

2.3.2 B-spline Network with Python Scikit Learn Library

Scikit-learn provides wide functionality and specialized packages for machine learning in Python. The use of those packages makes it possible to analyze data in a simple an efficient way. The packages can be used in various contexts and builds upon NumPy, SciPy and Matplotlib, Scikit-learn (2017).

2.3.2.1 SciPy

SciPy is open-source software for science, mathematics and engineering, Scipy Manual (2017). It is a collection of mathematical algorithms and functions that is built on Pythons extension Numpy. One of the sub-packages in NumPy is `interpolate` which consists of all kinds of interpolation functions and methods. From the interpolation package the functions `splrep` and `splev` are used to implement 1-dimensional B-spline networks and the functions `bisplrep` and `bisplev` are used for 2-dimensional networks.

The function `interpolate.splrep`, Splrep (2017) determines a smooth B-spline approximation of degree k on the interval $xb \leq x \leq xe$ given a set of data points $(x[i], y[i])$ defining the

curve $y = f(x)$. The function returns the 3-tuple `(t,c,k)` containing a knotvector, B-spline coefficients and the degree of the spline.

Important to note is that the supplied x data must be unique and the array content must contain the values in ascending order. Non-unique items should be filtered out before applying data to this function. Furthermore, the knots t must satisfy the Schoenberg-Whitney conditions, i.e. there must be a subset of data points $x[j]$ such that:

$$t[j] < x[j] < t[j+k+1] \qquad for\, j = 0, 1, .., n-k-2 \tag{2.10}$$

with, $t[j]$ knot at sample j
$x[j]$ input at sample j
k degree of B-splines
n number of samples

In words, Equation 2.10 tells that in between two consecutive knots of the knotvector a data point must exist.

The function `interpolate.splev`, Splev (2017) evaluates for some given input $x[i]$ the output value $y[i]$. In order to evaluate the data the 3-tuple `(t,c,k)` (the return from `interpolate.splrep()`) and the evaluation degree must be supplied. For input values being outside the defined interval of the knot sequence the returned output will be the extrapolated value by default, but it is possible to change this to return a 0, to return the boundary value or to raise an error.

The function `interpolate.bisplrep`, Bis (2017b) finds the bivariate B-spline representation of a surface. Data points for $x[i], y[i]$ and $z[i]$ are supplied that describe the surface $z = f(x, y)$. By supplying the knotvectors tx and ty (optional input) together with a certain B-spline degree the function returns a 5-tuple `(tx,ty,c,kx,ky)` that contains the knotvectors and the degree of the x- and y-dimension and one set of computed coefficients c. Optional parameters can be set to define the end points of the approximation interval for both x and y.

The function `interpolate.bisplev`, Bis (2017a) evaluates a bivariate B-spline (and its derivatives). The only compulsory inputs are the parameters that define the domain over which the spline has to be evaluated x, y and the 5-tuple `(tx,ty,c,kx,ky)` (returned from `bisplrep`). The return of the function (evaluation) is the cross-product of x and y. Initially the evaluation orders of x and y are set to zero, but those can be changed by defining dx and dy.

2.4 Illustrative Application: Linear Motor Motion System

The illustrative example used for the thesis is a model of a linear motor motion system. This type of motor is interesting in learning control and is widely used to perform linear motions that require sub-millimeter accuracy (i.e. scanning, laser cutting or pick-and-place tasks), Otten et al. (1997).

2.4.1 Introduction to a Linear Motor

The configuration of a linear motor consists of a base plate covered with permanent magnets and a translator that holds the electric coils with its iron cores. The translator undergoes the translational motions. In Figure 2.15 the working principle of a linear motor is illustrated.

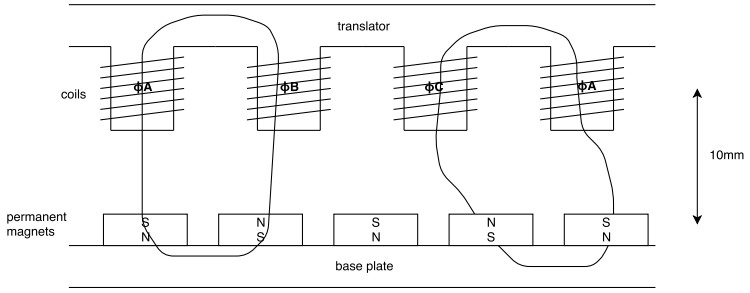


Figure 2.15: Working principle of a linear motor. The lines indicate the flux-lines of the permanent magnets and ϕ_a, ϕ_b and ϕ_c indicate the phases of the 3-phase motor current

The motion is established by applying a three-phase current to three adjoining translator coils. As a result, a series of attraction and rejection forces between the permanent magnets and the coils is generated. The basic behavior of the motor can be seen as the movement of a mass. For the thesis it is assumed that the total mass of the motor including a dummy load defined by $m_L = 37$ [kg].

The translator of the linear motor experiences a force ripple (disturbance) during its operation. The force ripple can be explained by two phenomena that occur:

- **Phenomena 1: Cogging force**
  Between the permanent magnets at the base plate and the translators iron cores a strong magnetic interaction takes place. Disturbance forces that try to align the magnets with the cores into a stable position of the translator cause these interactions. This force is called the cogging force and is independent of the motor current. It depends on the relative position of the translator with the magnets and is even present the moment the motor current is zero. A simplistic model to represent the cogging force F_C [N] is:

  $$F_C(x) = 10\sin(1.6 \cdot 10^{-2} x) \tag{2.11}$$

  The equation describes a sinusoidal shaped input disturbance that depends on the motor position x, has an amplitude of 10 [N] and a pitch of 0.016 [m]. In Figure 2.16 a measurement of the cogging of the real motor being modeled is shown.

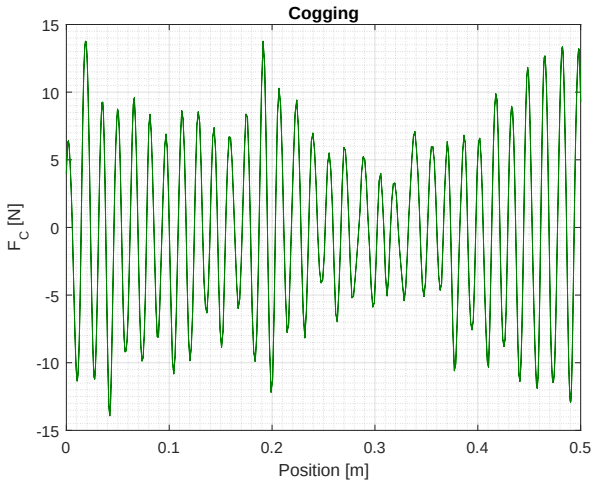


Figure 2.16: Input-output mapping of the position dependent cogging force, F_C

- **Phenomena 2: Back EMF**
  By commutation in the coils, i.e. the way the current is supplied to the coils, a force ripple

Table 2.1: System requirements of the feedback control system (PD controller plus moving mass)

Parameter	Value	Unit	Description
m_L	37	kg	Total moving mass (linear motor plus dummy load)
$\ddot{x}_{max}$	10	m/s^2	Maximum acceleration of the mass
$e_{max,track}$	100	μm	Maximum tracking error of the control system

can be generated. Back EMF is generated the moment a coil is moved through a varying electro-magnetic field. If the current supplied to the coils is not proportional to the back EMF, a force ripple will appear.

Using a detailed model of the structure of the motor enables the computation of the back EMF, but in order to do so accurate data about the position and magnetic properties of the permanent magnets is required. In most applications linear motors are used that have large magnetic tolerances (beneficial to reduce the motor costs). For each individual motor the model should be individually adjusted which makes implementing them more difficult. Therefor it is decided to omit the force ripple caused by this fact.

Besides the cogging force, friction and motor inertia are commonly taken into account in modeling a linear motor. An example of a friction force that is encountered is the moment the translator of the motor slides along the guiding rails. The simulations performed for this thesis only incorporate the inertia of the mass (together with the cogging force). The friction will be omitted as this force is negligible when compared to the inertia.

2.4.2 Design of Linear Motor Model

The model shown in Figure 2.17 is used in simulations to model the linear motor as a plant. It includes the inertia of the mass of the linear motor and the position dependent cogging.

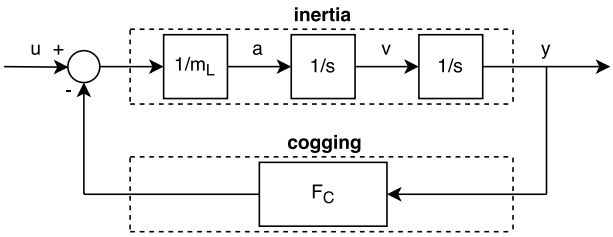


Figure 2.17: Plant model for the non-ideal linear motor in which the inertia of mass m_L and the cogging are included

2.4.3 Design of Feedback Controller

The feedback controller compensates for random disturbances and generates the learning signal (target) for the LFFC. Tuning a feedback controller is based on certain plant requirements, therefor values are assumed for the maximum allowable tracking error $e_{max,track}$, the maximum possible acceleration $\ddot{x}_{max}$ of the mass and the total mass m_L to be displaced, see Table 2.1. An ideal model of a moving mass is assumed for tuning the PD controller.

The feedback controller is formed by a combination of a proportional (P) and a derivative (D) action. The P-action produces a control action proportional to the produced error. The larger the error the larger the controller output. The D-action produces a control action proportional to the rate of change of the error. The more sudden the error changes the larger the control output.

It is chosen to represent the PD controller in serial form as this allows for better tuning in the frequency domain. de Vries (2015) provides the information to design a serial PD controller in transfer function form. The design consists of a combination of the selection of controller gain K_C and a filter formed by specific pole-zero placement of τ_z and τ_p.

$$C_{FB} = K_C \cdot \frac{s\tau_z + 1}{s\tau_p + 1} \tag{2.12}$$

The design starts with defining the cross-over frequency from the maximum acceleration of the mass $\ddot{x}_{mass}$, the maximum allowable tracking error $e_{max,track}$ and the tameness factor β (as a rule of thumb, $\beta = 10$):

$$\omega_c = \sqrt{\frac{\ddot{x}_{max} \cdot \sqrt{\beta}}{e_{max,track}}} \tag{2.13}$$

From Equation 2.13 the controller gain K_C, τ_z and τ_p can be determined:

$$K_C = \frac{m_L \omega_c^2}{\sqrt{\beta}} \tag{2.14a}$$

$$\tau_z = \sqrt{\beta} \cdot \frac{1}{\omega_c} \tag{2.14b}$$

$$\tau_p = \frac{1}{\sqrt{\beta} \cdot \omega_c} \tag{2.14c}$$

The systems cut-off frequency is found to be 562 [rad/s] and the controller transfer function is:

$$C_{FB} = 3.7 \cdot 10^6 \cdot \frac{5.622 \cdot 10^{-3} s + 1}{5.623 \cdot 10^{-4} s + 1} \tag{2.15}$$

2.4.4 Performance Check on the Feedback System Model

The performance of the designed feedback controller is checked by implementing the feedback controlled system in the simulation software 20-sim. The model is shown in Figure 2.18.

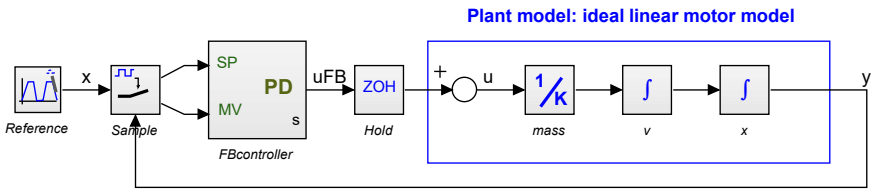


Figure 2.18: 20-sim model to check performance of control system

A "MotionProfile" supplies the control system of a partial cubic reference signal having a maximum acceleration of 10 [m/s^2] and a maximum displacement of $x_{max} = 0.5$ [m], the complete set of signal parameters is shown in Table 2.2. In Appendix A a description of the partial cubic signal together with its parameter definitions are shown. The feedback controller used is discrete such that the option is enabled to control a linear motor that is outside the simulation environment (a real world set-up). Though, for this thesis a simulation model of the linear motor is used only.

Table 2.2: Reference signal parameters for performance check of tuned PD-controller

Parameter	Value	Unit	Description
$rise\_time$	0.527	s	Rise time
$start\_time$	1.000	s	Start time
$stop\_time$	1.527	s	Stop time
$return\_time$	2.000	s	Return time
end_time	2.527	s	End time
$period$	3.527	s	Period of signal
j_{max}	189.737	m/s^3	Maximum jerk
a_{max}	10.000	m/s^2	Maximum acceleration
v_{max}	1.581	m/s	Maximum velocity
x_{max}	0.500	m	Maximum displacement (= stroke)
CV	20	%	Percentage Constant Velocity (CV)
CA	20	%	Maximum Constant Acceleration (CA)

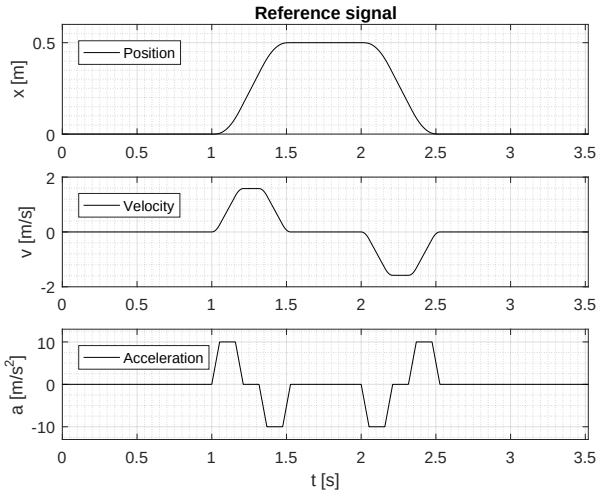


Figure 2.19: Reference signal used to check the performance of the tuned PD-controller

The maximum allowable tracking error was defined to be 100 [μm] and from Figure 2.20 it can be observed that this requirement is met.

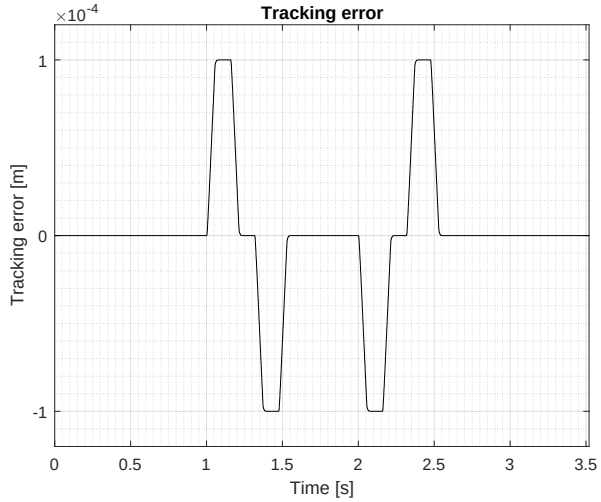


Figure 2.20: The controlled systems tracking error obtained after tuning the PD-controller

3 Network Communication

3.1 Introduction

A feedback control system can be extended with a feed-forward controller. The most convenient way to obtain this is by just implementing both controllers on the same platform, for instance both in the simulation environment of 20-sim. For the thesis the learning controller (feed-forward part) will be implemented in Python. By setting up a network connection between the feed-forward controller in Python and (in this case) the feedback system plus plant in the simulation software 20-sim both can communicate.

The network part is set-up by making use of ZeroMQ, ZeroMQ (2017) and of Protocol Buffer from Google, ProtoBuf (2017). The combination of both enables a well structured data communication structure between 20-sim and Python. In this chapter the design of the network layer is presented and its implemented is shown. Two tests are performed to verify if the communication works as expected.

3.2 Design of Network Layer

A general feedback control system that is extended with a properly set feed-forward controller shows improved performance. Though it requires both parts to be used on the same computing platform (i.e. Linux, Microsoft Windows or macOS) and all parts needs to be operated in the same operating environment (for instance a simulation environment). To abolish these restrictions a network layer is added in between the feedback part and the feed-forward part. As a result, both parts can be implemented on different computing platforms. Communication remains possible by setting up a network link. In Figure 3.1 a diagram is shown in which a network layer is included within a learning feed-forward controlled system.

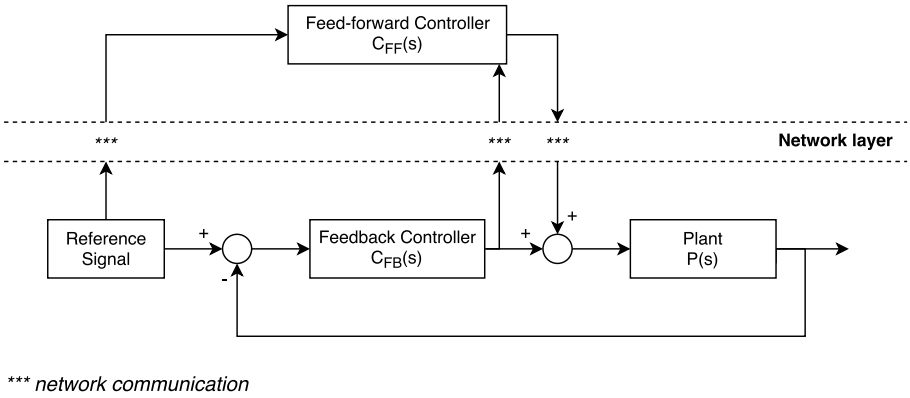


Figure 3.1: Learning feed-forward controlled system that includes a network layer

The feed-forward controller is implemented in Python and contains a function approximator from the Python Scikit learn library. The input-output mapping of the system is adapted during control in order to behave as desired. The feedback part (in this experiment) is implemented in the simulation environment of 20-sim.

The feed-forward controller has to perform three tasks:

1. Collect data
2. Approximate behavior of the inverse plant, $P^{-1}(s)$
3. Evaluate the inverse plant approximation and apply to the system

The tasks the feed-forward controller needs to perform do not all require network communication. Only tasks 1 and 3 have to communicate over the network, see Figure 3.2.

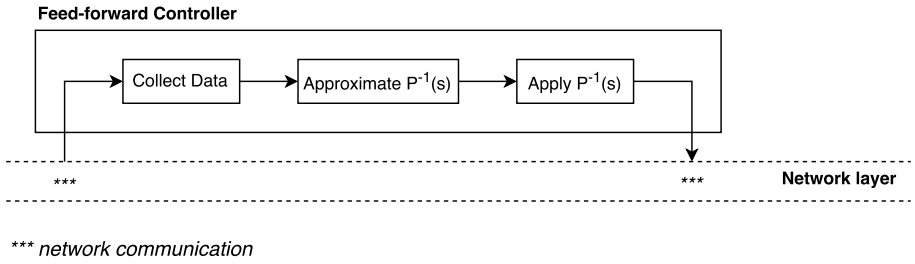


Figure 3.2: Tasks of the feed-forward controller, (***) indicates a task requiring network communication

In order to obtain a safe and reliable communication between both sides of the network layer, two protocols are used: ZeroMQ and Protocol Buffer from Google. The latter packs the data into a serialized string such that the data can be sent and received according to one and the same format. The former sends the serialized string over the network. ZeroMQ can be seen as a concurrency framework and is already prepared to carry messages along various transport processes (inter-process, in-process, TCP and multicast). The network communication channel is set-up by binding two IP-addresses to each other. At both sides of the network the IP-address of the application containing the feed-forward part must be available.

3.3 Implementation of Network Layer

ZeroMQ, ZeroMQ (2017) and Protocol Buffer, ProtoBuf (2017) from Google are used together to make it possible to transfer data from 20-sim to Python and vice versa. In this section some more information is provided about both protocols.

3.3.1 ZeroMQ

ZeroMQ can be seen as a concurrency framework that provides sockets to carry messages across various transports, i.e. inter-process, in-process, TCP and multi-casts. It can connect N-to-N sockets according to several patterns such as fan-out, pub-sub and request-reply. The latter is used to bind both applications together and a connection between client (20-sim) and server (Python) needs to be set-up manually. This is done by providing the IP-address of the server to the client application, which must be contained in the config file (`.cfg`) that will be automatically read as soon a simulation is started.

The communication pattern starts with a request (REQ) sent by the client using `zmq_send()`, the server (Python) receives the request and performs a calculation followed by sending a reply (REP) back to the client, which is received using `zmq_recv()`. This way 20-sim can send data to Python (for data storage and processing) and Python can send data (processed data) back to 20-sim to be used in the simulation. In Figure 3.3 a diagram is shown in which the communication between client and server is visualized.

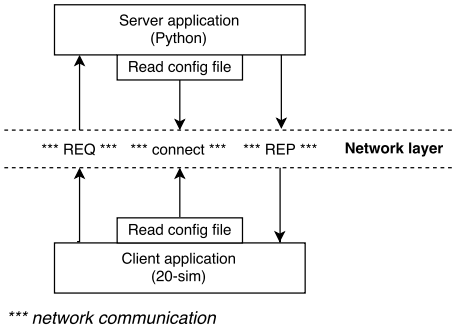


Figure 3.3: ZeroMQ communication diagram between client and server. Three network communication actions in the order 1) connect, 2) client request (REQ) and 3) server reply (REP)

3.3.2 Protocol Buffer from Google

Protocol Buffer from Google is used to serialize data in a structured way (creating a message) with its property of being language and platform neutral. The structure of the data only has to be defined once and after that the special generated source code can be used to write and read data from the structure using variety of languages and from variety of data streams.

The data structure of the message is specified by defining a protocol buffer message of the type `.proto`. The message is small and forms a logical record of information contained in a series of name-value pairs. Each message type has one (or more) uniquely numbered fields defined by a name and a value type. Possible value types are for instance booleans, numbers, raw bytes and strings. The fields can be specified as optional, required and repeated fields as well.

In the control systems client-server setting, before the client actually sends out its request (REQ) the data is packed into a message (containing a serialized string) by Protocol Buffer. The server receives the request and unpacks the message using Protocol Buffer such that actions can be performed on the data. The new data is packed by Protocol Buffer into a message which is send as a reply (REP) back to the client. The client receives the reply, unpacks the data again with Protocol Buffer. The data is ready to be used by 20-sim. The packaging structure is illustrated in Figure 3.4.

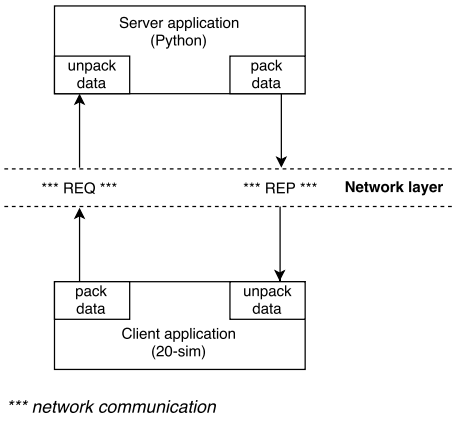


Figure 3.4: Diagram of the data (de-)serialization process of Protocol Buffer from Google in combination with network communication between client and server

3.3.3 Network Communication with 20-sim

The feedback control system is implemented in the simulation environment of 20-sim. Since it is not possible to directly communicate with 20-sim from an external application a DLL is used to make this possible. A DLL is a Dynamic Link Library and is a file that contains instructions that can be called by another program to perform a certain task (for instance tasks requested by 20-sim). It is not possible to run a DLL file directly (like an executable file, `.exe`) but it must be called by some other running code. The term 'dynamic' refers to the fact that the data within the file is only used the moment a program actively calls the data. As a result, the data is not always available in memory. In order to run simulations in 20-sim that include network communication, the program is required to perform a DLL call to a `.dll` (created by a build of code written in C++) such that via that file a temporarily communication link is opened to the Python server, and vice versa. A DLL call in 20-sim can be done as follows:

```
output=dll(dll_name,function_name,input)
```
(3.1)

with, `dll_name`: the name of the DLL file with extension `.dll`
`function_name`: the name of the function containing the action to be performed
`input`: input data to be passed on the function that is called

This function returns data that is configured as `output`. It is possible to perform calls that pass on single or multiple inputs and/or that receive single or multiple outputs. The diagram in Figure 3.5 visualizes a DLL call between the 20-sim client and the Python server.

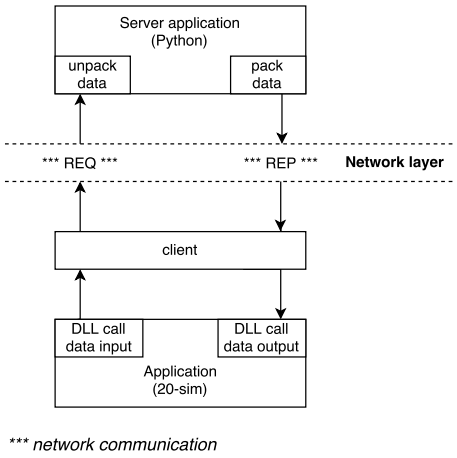


Figure 3.5: Diagram of DLL call from 20-sim to Python server

The 20-sim simulation software supplies the user with a standard structure for making a DLL call. A Visual C++ code example is provided 20simDynamicDLL (2017) which can be used and modified to achieve specific functionality. In considering a multiple run simulation experiment the actions that are consecutively performed by the simulation are:

1. `Initialize()`
   Function called by 20-sim before the simulation experiment is started, and only once in a multiple run experiment. Initializations of data structures can be placed within this function.

2. `InitializeRun()`
   Function called by 20-sim before a simulation experiment starts in a multiple run experiment.

3. `dllFunction()`
   Function called during a simulation experiment in which for instance data can be captured and modified.

4. `TerminateRun()`
   Function called by 20-sim after each finished run. For instance, clean-up of data can be done within this function.

5. `Terminate()`
   Function called by 20-sim after a finished (multiple run) simulation experiment.

The above structure will be used to create an efficient communication between the 20-sim client (making a DLL call) and the Python server. This structure will ensure that actions only take place the moment they need to.

3.4 Validation of Network Layer

The proposed network communication set-up is evaluated by performing two tests. One to see if a server (in Python) and a client (in C++) can communicate with each other by sending each other a string. The second test will verify if 20-sim can make a DLL call to a `.dll`-file that was created by building a program written in C++. For this test a simulation in 20-sim is started and data is send to the Python server. The server manipulates the received data and send it back to the client.

For both tests two devices (laptops) are connected to the same network and both IP-addresses are within the same subnet. This way it is enabled that both devices are allowed "to see" each other and the possibility arises to set-up a connection between them. The IP-addresses of the client and server are `IPv4:192.168.0199` and `IPv4:192.168.0.200`. At both applications the IP-address of the server needs to be known. At the server side, the IP-address used is referred to by using local host and at the client side a `.cnf` (config) file is read out in which the IP-address of the server needs to be inserted manually.

3.4.1 ZeroMQ Test

The "ZeroMQ test" is a test to validate if it is indeed possible to let a client implemented in C++ communicate over the network to a server implemented in Python. The client will send the string "Hello" and it is intended to receive the string "World" back from the server. The ZeroMQ connection is based on the Request-Reply (REQ-REP) mode. To validate if it works, the server in Python `ZeroMQ_server.py` is started followed by executing the client `testZeroMQ.exe`. The output of the server is shown in the "IPython console" and the output of the client appears in the command prompt window, see Table 3.1.

Table 3.1: Ipython console output and command prompt window output, after completed communication between client and server

testZeroMQ_server.py	testZeroMQ.exe
Received request: b'Hello'	Reading IPaddress from file... Connecting to hello world server ... Sending Hello Received World

Furthermore, the software *Wireshark* is used to track the network communication of both applications, see Figure 3.6. Multiple packages go from one side to the other. Row 9 and 11 contain the string "Hello" and "World", see Table 3.2 for more detailed information about the content of the packages.

((ip.src==192.168.0.200) && (ip.dst == 192.168.0.199)) or ((ip.src==192.168.0.199) && (ip.dst == 192.168.0.200))

No.	Time	Source	Destination	Protocol	Length	Info
…	80.591164	192.168.0.199	192.168.0.200	TCP	66	59528 → 5555 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
…	80.594478	192.168.0.200	192.168.0.199	TCP	66	5555 → 59528 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
…	80.594581	192.168.0.199	192.168.0.200	TCP	54	59528 → 5555 [ACK] Seq=1 Ack=1 Win=65536 Len=0
…	80.595103	192.168.0.199	192.168.0.200	TCP	64	59528 → 5555 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=10 [TCP segment of a reassembled PDU]
…	80.596550	192.168.0.200	192.168.0.199	TCP	65	5555 → 59528 [PSH, ACK] Seq=1 Ack=11 Win=65536 Len=11 [TCP segment of a reassembled PDU]
…	80.596812	192.168.0.199	192.168.0.200	TCP	108	59528 → 5555 [PSH, ACK] Seq=11 Ack=12 Win=65536 Len=54 [TCP segment of a reassembled PDU]
…	80.598597	192.168.0.200	192.168.0.199	TCP	107	5555 → 59528 [PSH, ACK] Seq=12 Ack=65 Win=65536 Len=53 [TCP segment of a reassembled PDU]
…	80.598598	192.168.0.200	192.168.0.199	TCP	81	5555 → 59528 [PSH, ACK] Seq=65 Ack=65 Win=65536 Len=27 [TCP segment of a reassembled PDU]
…	80.598668	192.168.0.199	192.168.0.200	TCP	54	59528 → 5555 [ACK] Seq=65 Ack=92 Win=65536 Len=0
…	80.599052	192.168.0.199	192.168.0.200	TCP	103	59528 → 5555 [PSH, ACK] Seq=65 Ack=92 Win=65536 Len=49 [TCP segment of a reassembled PDU]
…	80.803998	192.168.0.200	192.168.0.199	TCP	60	5555 → 59528 [ACK] Seq=92 Ack=114 Win=65536 Len=0
…	81.615146	192.168.0.200	192.168.0.199	TCP	63	5555 → 59528 [PSH, ACK] Seq=92 Ack=114 Win=65536 Len=9 [TCP segment of a reassembled PDU]
…	81.616207	192.168.0.199	192.168.0.200	TCP	54	59528 → 5555 [FIN, ACK] Seq=114 Ack=101 Win=65536 Len=0
…	81.617578	192.168.0.200	192.168.0.199	TCP	60	5555 → 59528 [ACK] Seq=101 Ack=115 Win=65536 Len=0
…	81.617579	192.168.0.200	192.168.0.199	TCP	60	5555 → 59528 [FIN, ACK] Seq=101 Ack=115 Win=65536 Len=0
…	81.617637	192.168.0.199	192.168.0.200	TCP	54	59528 → 5555 [ACK] Seq=115 Ack=102 Win=65536 Len=0

Figure 3.6: *Wireshark* package flow for communication between client and server

Table 3.2: Part of the package content containing "Hello" and "World", data extracted using *Wireshark*

Source IP	Destination IP	Information
`192.168.0.199`	`192.168.0.200`	`[PSH, ACK] Seq=65 Ack=92 Win=65536 Len=49 [TCP segment of reassembled PDU] TCP payload (49 bytes): containing Hello`
`192.168.0.200`	`192.168.0.199`	`[PSH, ACK] Seq=114 Ack=144 Win=65536 Len=9 [TCP segment of reassembled PDU] TCP payload (9 bytes): containing World`

3.4.2 Data Transfer Test

The "data transfer" test is performed to see if the simulation software (20-sim) can communicate with a server in Python, by performing a DLL call. 20-sim makes a call to the function `dataTransfer`, a request (from the client) is send to the server and a reply is received back from the server.

A 20-sim simulation is started in which a reference input signal $2\sin(t)$ is used, $t \in [0, 2\pi]$. The block "DLLcall" contains the function to make a DLL call (using 3.1) such that a communication link between the Python server and the client is established. Input for the function are samples of the reference input and the time obtained by using a sample time of 0.01 $[s]$. The purpose of the experiment is that Python receives the sampled data, multiplies the references signal by two and sends back the result to 20-sim. The simulation model used is shown in Figure 3.7.

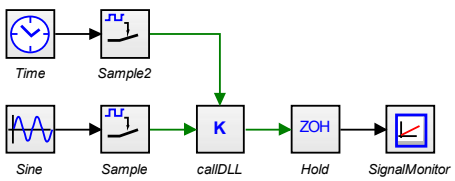


Figure 3.7: 20-sim model used to perform the data transfer test for communication between 20-sim and Python

The data is saved at both ends of the network link (at the 20-sim side and at the Python side) and it is expected that the data of the out- and input of 20-sim represents the functions $2\sin(t)$ and $4\sin(t)$. At the Python side it is expected that the in- and output represent the functions $2\sin(t)$ and $4\sin(t)$, the results are shown in Figure 3.8. From observations it can be concluded that the correct data is send/received at both ends of the links such that the "data transfer" test is succeeded.

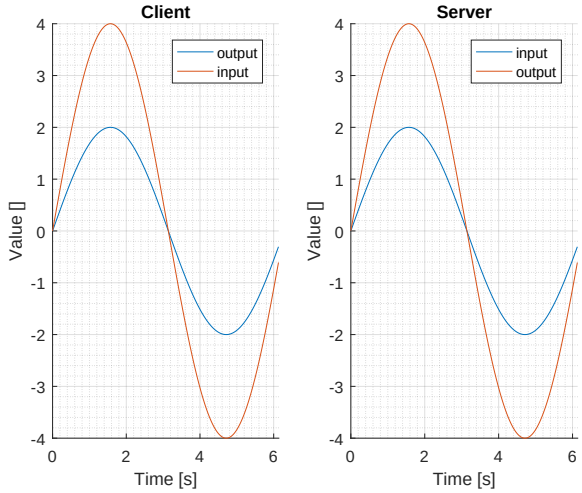


Figure 3.8: In- and output data plotted at a) client side (20-sim) and b) server side (Python)

3.5 Conclusion

The client (C++) and server (Python) in the "ZeroMQ test" correctly communicate, the client sends the string "Hello" to the server, the server receives it and sends "World" back to the client (and receives it).

The "data transfer" test showed that 20-sim was able to correctly communicate to the server by a DLL-call . To do so, a simulation was started in which an input signal ($2\sin(t)$) was generated, sampled and send to the server.

It can be concluded that the combination of ZeroMQ and Protocol Buffer from Google can be used to set-up a network communication link between 20-sim and Python.

Although not demonstrated in this chapter, the server could have run on a Linux platform. As, Python, Protocol Buffer from Google and ZeroMQ can run on the Linux operating system, without any modifications.

4 One Dimensional LFFC

A feedback control system that is extended with a properly set 1-dimensional (1D) learning feed-forward controller shows improved system performance. Depending on the input signal supplied to the learning controller it is suited for repetitive motions or non-repetitive motions. A time-indexed LFFC (input is a function of the periodic motion time) is used for repetitive motions and a state-indexed LFFC (input is the reference position, velocity or acceleration) is used for non-repetitive motions.

This chapter starts with the design of two 1-dimensional time-indexed learning feed-forward controllers for which the environment in which they are implemented differ. For both implementations the reference signal is produced by the built-in "waveform generator" of 20-sim and the signals shape it produces is referred to as a partial cubic motion. A general feedback system is used in which a tuned PD-controller controls the ideal plant model representing a moving mass.

The feed-forward part contains a learning controller that will approximate the mass of the plant by using a B-spline Network. The difference between both time-indexed LFFCs is found within this part. One implementation uses the built-in B-spline Network editor of 20-sim and the other uses the Scikit-learn library from Python. The latter implementation requires the set-up of a network connection between 20-sim and Python, for more details see Chapter 3.

Both implementations are compared by evaluating the simulation results obtained by 20-sim, but also the user friendliness and the design freedom during those simulations is taken into account. A time-indexed LFFC is implemented using Python only and the task of the learning controller is to learn the inverse behaviour of the plant, i.e. its mass.

The designed time- and state-indexed LFFCs have the B-spline networks designed using parameters which seem to cause no instable behavior. Though, it might be possible that for instance by extending the simulation periods and/or the number of runs (within a multiple run session) unstable behavior will occur. Designing a perfect LFFC is beyond the scope of this thesis.

4.1 Time-Indexed LFFC

Time-indexed learning feed-forward controllers are beneficial for linear motor motion systems that has to perform repetitive motions. This type of motions will observe at each time instance the same plant influences and other disturbances each time the motion repeats. Therefor, the input supplied to the learning controller can be chosen to be a function of the periodic motion time, T_p for which the motion is defined on the input domain $t \in [0, T_p]$. The structure of a time-indexed LFFC is shown in Figure 4.1.

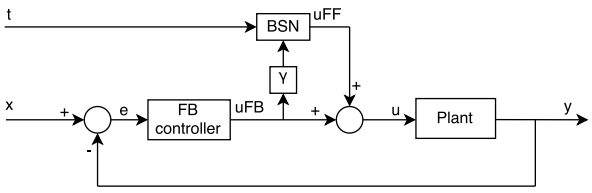


Figure 4.1: Time-indexed LFFC structure (1-dimensional)

4.1.1 Design

The 1-dimensional time-indexed LFFC is designed using a B-spline network (BSN) as its function approximator, therefor the BSN network parameters must be selected. The approximator

Table 4.1: Trainings signal parameters time-indexed LFFC

Parameter	Value	Unit	Description
T_{rise}	0.786	s	Rise time
T_{start}	1.000	s	Start time
T_{stop}	1.786	s	Stop time
T_{return}	2.000	s	Return time
T_{end}	2.786	s	End time
T_p	3.786	s	Period of signal
j_{max}	57.276	m/s^3	Maximum jerk
a_{max}	4.500	m/s^2	Maximum acceleration
v_{max}	1.061	m/s	Maximum velocity
x_{max}	0.500	m	Maximum displacement (= stroke)
CV	20	%	Percentage Constant Velocity (CV)
CA	20	%	Percentage Constant Acceleration (CA)

has to learn the inverse behavior of the ideal plant for which a model is used that incorporates the inertia of the mass m_L only. A more detailed description of the plant model is given in Section 2.4 and the model itself is shown in Figure 4.2.

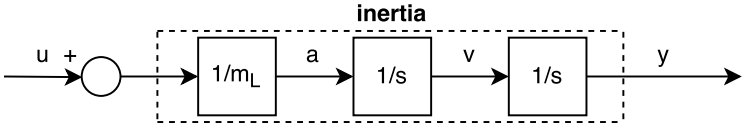


Figure 4.2: Ideal plant model of linear motor

The parameters to be selected are the upper and lower input values, the learning rate, the number of B-splines, the distribution of the B-splines and the degree of the B-splines. Especially the number of B-splines and the learning rate are important to select properly, because those parameters influence the stability of the learning controller. The B-spline network setting are determined according to the following step-by-step plan:

Step 1: Input selection of the BSN
The input selection for the BSN is based on the motions to be performed by the linear motor. Since a time-indexed LFFC shows only good performance for repetitive motions input t is selected for the learning controller, which is a function of the periodic motion time T_P.

Step 2: Selection of the B-spline order
Along with the degree of the B-splines the smoothness and accuracy of the function approximation is set. The higher the degree the more complex the computation of the B-splines and therewith the longer the process time will be. First degree (or second order) B-splines will be used.

In using this type of basis functions a smooth enough approximation can be achieved while limiting the computational complexity, as suggested in Velthuis (2000). First degree B-splines are described by two linear line segments forming a triangular shape, see Section 2.2.1.

Step 3: Selection of the trainings motion
A partial cubic motion profile is used as trainings signal as described in Appendix A. The acceleration is limited to 4.5 [m/s^2] and a maximum displacement is 0.5 [m] (forward and backward motion). The signal is shown in Figure 4.3 and its signal parameters in Table 4.1. The motion shows start and end periods in which no motion is performed such that unwanted behavior can be observed in those areas.

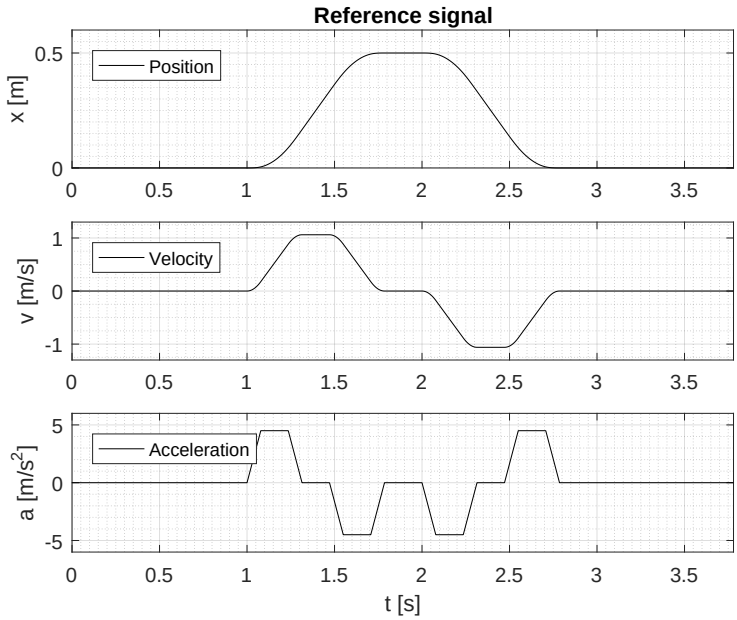


Figure 4.3: Trainings motion for the time-indexed LFFC (acceleration) and derivatives thereof

Step 4: Selection of the B-spline distribution and the number of splines
The distribution of the B-splines will be uniformly, which means that each B-spline is equally spaced over the input domain from $[0, T_P]$. Reason for this is that it is more easily to implement while still sufficient simulation results can be obtained. Besides that, in 20-sim it is not possible to define a non-uniform B-spline distribution.

In most situations an exact model of the plant is not available. In order to be able to still determine the minimum B-spline width of the network the step-by-step plan given in "Algorithm 2.3", Velthuis (2000) can be used:

4.1: **Find infinity norm of the inverse complementary sensitivity function**
The infinity norm is defined as $\left|-T(j\omega)\right|_\infty$. The inverse complementary sensitivity function $-T(j\omega)$ (closed system) is used to determine this norm. In Figure the diagram is shown from which is found that:

$$\left|-T(j\omega)\right|_\infty = 1.275[dB] \tag{4.1}$$

4.2/4.3: **Find the minimum frequency for which the cosine of the Phase (Step 4.1) is smaller or equal than zero**
In Figure 4.5 $\cos(\phi)$ is shown in which ϕ represents the phase obtained from $\left|-T(j\omega)\right|_\infty$ (shown in Figure 4.4). The minimum frequency for which $\cos(\phi) \leq 0$ is:

$$\min_{\omega\in\mathbb{R}|\cos(\phi\leq 0)} \left|-T(j\omega)\right| \rightarrow \omega = 852.699[rad/s] \tag{4.2}$$

The smallest ω_1 at which $\phi_1 = \arg(-T(j\omega_1))$ satisfies $\phi_1 = \arccos\left(-0.0147 \frac{\left|-T(j\omega)\right|_\infty}{\min_{\omega\in\mathbb{R}|\cos(\phi\leq 0)}\left|-T(j\omega)\right|}\right)$ is found using Step 4.1 and Step 4.2:

$$\begin{aligned}\phi_1 &= \arccos\left(-0.0147 \frac{\left|-T(j\omega)\right|_\infty}{\min_{\omega\in\mathbb{R}|\cos(\phi\leq 0)}\left|-T(j\omega)\right|}\right) \\ &= \left(-0.0147 \cdot \frac{1.275\ [\text{dB}]}{1.122\ [\text{dB}]}\right) \\ &= 91.074\ [\text{deg}]\end{aligned} \tag{4.3}$$

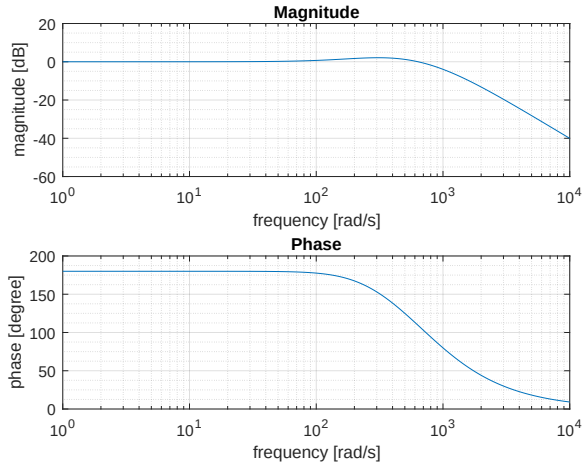


Figure 4.4: Bode diagram of the inverse complementary sensitivity function, i,e. $\left|-T(j\omega)\right|_\infty$ (from the closed loop system)

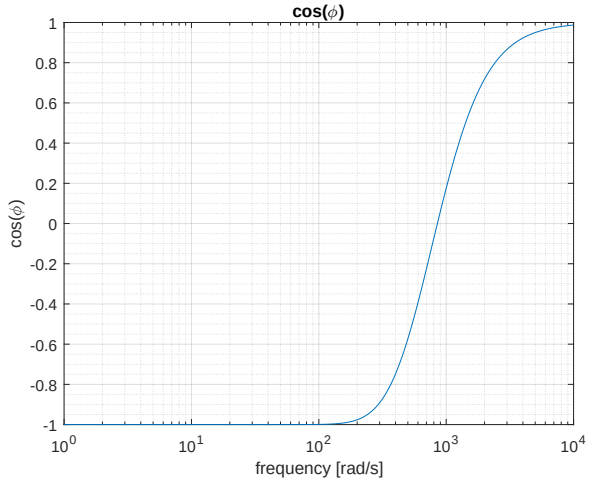


Figure 4.5: Plot of the cosine of the phase observed from $-T(j\omega)$

4.4: **Find the minimum B-spline width d_{min}**
Using the obtained ω_1 (from Step 4.2/4.3) the minimum width d_{min} of the B-splines is determined according to:

$$d_{min} = \frac{2\pi}{\omega_1} \quad \rightarrow \quad d_{min} = \frac{2\pi}{852.699} = 7.369 \cdot 10^{-03} \tag{4.4}$$

By dividing the motion period $T_p = 3.78$ [s] by d_{min} the maximum number of B-splines is obtained:

$$N_{max} = \frac{T_p}{d_{min}} \quad \rightarrow \quad N_{max} = \frac{3.78}{7.369 \cdot 10^{-03}} = 512 \tag{4.5}$$

The number of splines must be lower than 512. In order to compare the BSN implementations of 20-sim and python two values are chosen, i.e. 400 splines and 500 splines.

In Figure 4.6 the uniform distribution of N B-splines of second order (degree zero) is shown. The basis functions are sequentially labeled by $i \in 1...N$. The distribution shows that the first (1) and last (N) basis function have half the width of the basis functions at the center part. This type of learning controller is used for performed motions that are completely independent. This means that before a new motion is started, the system is brought back to its initial states. A typical application in which this type of "reset" is used is a pick-and-place machine.

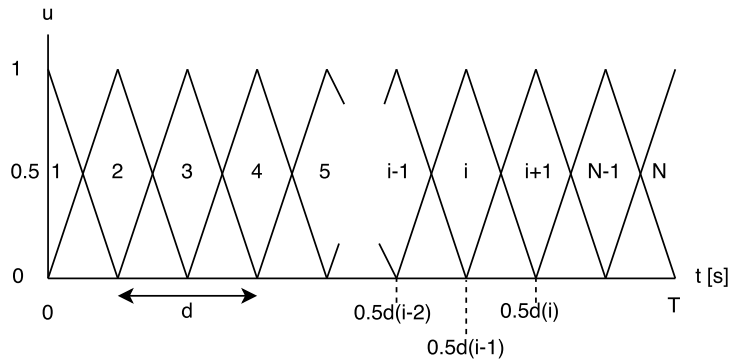


Figure 4.6: A B-spline distribution of time-indexed LFFC

Step 5: Selection of the learning rate
The learning rate of the feed-forward controller is wished to be as large as possible such that convergence is reached fast. Though, too high rates may result instable behavior. In Velthuis (2000) an equation is proposed in order to determine the maximum possible learning rate:

$$\gamma \leq \frac{2}{\left|-T(j\omega)\right|_\infty} \quad \rightarrow \gamma \leq 0.948 \tag{4.6}$$

For the simulations the learning rates are set to $\gamma = 0.5$ and $\gamma = 0.9$, both lower than the maximum allowable value found from Equation 4.6 to avoid unstable behavior.

4.1.2 Implementation

Two different implementations of the B-spline network of the feed-forward controller are considered. The first implementation is completely built within the simulation environment of 20-sim using the built-in B-spline Editor. The second implementation makes use of the Python Scikit-learn library. The latter implementation operates outside the 20-sim environment and therefor a network link is set-up between 20-sim and Python (see Chapter 3).

4.1.2.1 B-spline Network with 20-sim B-spline Editor

In this section the built-in B-spline network editor (from the software 20-sim) is used to implement the B-spline network in the feed-forward part of the control system. In Figure 4.7 the 20-sim model used for this is shown. The plant model only includes the influence of the inertia of the mass $m_L = 37$ [kg].

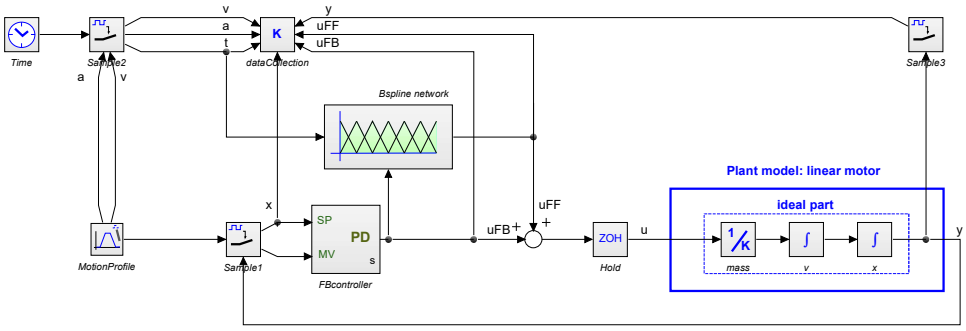


Figure 4.7: 20-sim model for time-indexed LFFC using 20-sim implementation with ideal plant

The built-in waveform generator of 20-sim is used to supply input x to the model, see Figure 4.3. The input for the B-spline network is t sampled each millisecond. Furthermore, the network is supplied with the so called target signal. This signal is the output of the feedback controller multiplied with the learning rate (γ) plus the feed-forward signal $u_{FFprevious}$ of the previous run (at a specific sample instance). The moment no previous data is available signal $u_{FFprevious} = 0$.

Depending on the observed tracking error (difference between the system output y and the desired output y_d) and the learning rate the B-spline network will adapt its network weights. The output of the B-spline network is the feed-forward control signal, u_{FF}, that is fed into the control system just before the input of the plant.

The function approximation is performed by the BSN in 20-sim and therefor the "dataCollection" block is used only to send the collected data over the network to the Python server. The server saves the data to file in (`.txt`) format. The collected data is the time t [s], reference inputs (x [m], $\dot{x} = v$ [m/s], $\ddot{x} = a$ [m/s^2]), feedback output u_{FB} [N], feed-forward output u_{FF} [N] and the system output y [m].

4.1.2.2 B-spline Network with Python Scikit-Learn

The model used to implement the Python B-spline network is shown in Figure 4.8. The plant model includes the inertia of the mass only.

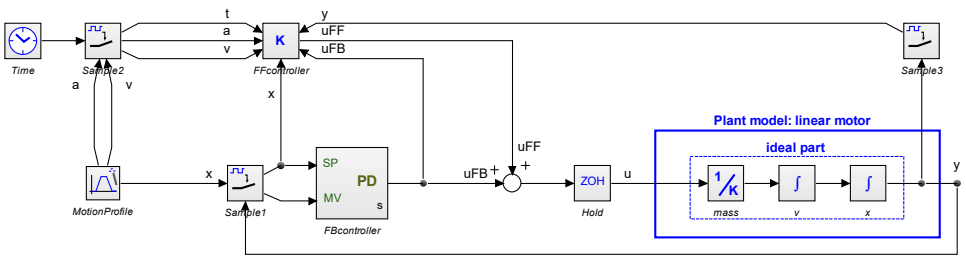


Figure 4.8: 20-sim model for time-indexed LFFC implemented with a Python B-spline network

A multiple run simulation experiment in 20-sim is assumed in which at each sample time a DLL-call is made to the python server. The server captures data, performs the function approx-

imation and send back the feed-forward control signal (for more detailed information about the DLL-call see Section 3.3.3). This functionality is wrapped in the block "FFcontroller". This block has six inputs, i.e. time t [s], reference input x [m] (and its derivatives $\dot{x}$ [m/s], $\ddot{x}$ [m/s^2]), feedback output u_{FB} [N] and the system output y [m]). Furthermore the block has one output: the feed-forward output u_{FF} [N]. All in- and outputs of the block are sampled instances at each 1 [ms].

The function approximation is based on the Python Scikit-learn library, see Section 2.3.2 and it uses the function `interpolate.splrep` to determine the network coefficients (adapted weights) and the function `interpolate.splev` to evaluate the output of the B-spline network. After a run is completed the adapted weights c are stored to file together with the knotvector t and the degree of the splines k to form the 3-tuple `(t,c,k)`. The next run this file is reloaded and used as the starting point for the function approximation.

The `splrep` has some requirements that must be fulfilled in order to perform a proper function approximation:

1. input data is unique (all input data has a multiplicity of one)
2. input data is sorted in ascending order
3. an input data point must be present within two consecutive knots of the knotvector

As the time-indexed LFFC uses time as its input the data is automatically unique and sorted in ascending order. The third requirement is fulfilled as long as the sample time is smaller than the number of knots (B-splines) used. Since the sample time is 0.001 [s] and the number of B-splines are selected to be either 400 or 500, this requirement is met. In order to perform proper function approximation it is not necessary to perform some kind of data preprocessing.

The process from start till end of a multiple run simulation experiment can be split-up into five stages, according to Section 3.3.3. The most important actions performed per stage are described in Table 4.2.

4.1.3 Comparison 20-sim and Python

In order to compare the B-spline Network implementation of 20-sim and Python, 20-sim simulations are performed such that the performances of both can be evaluated. Besides the simulations, both are compared based on the user friendliness and the freedom of the designer as well.

4.1.3.1 Simulations

The B-spline network implementations of python and 20-sim are compared based on the results of multiple run simulation experiments that each consists of 25 runs. The partial cubic motion pattern shown in Figure 4.3 is used.

In Python it is possible to set the degree of the output evaluation of the network while this is not possible in 20-sim. Nevertheless, this value is set to zero as for this thesis it is just wishful to use the general feed-forward signal (u_{FF}) in stead of a derivative thereof. A summary of the network settings is listed in Table 4.3.

The differences between the 20-sim and python implementation are studied by looking at the tracking error, the learned signals (i.e. u_{FF} vs. a) and the values the learning controllers converge to.

Table 4.4 lists the absolute maximum tracking error and the absolute mean tracking error for both the 20-sim and Python implementation. The values shown for "Run 1" describe the situation in which the learning controller hasn't learned yet. In fact, this can be seen as using a control system having a feedback system only. As a result, the values obtained are equal for 20-sim and Python and independent of the number of B-splines and the learning rate chosen.

Table 4.2: Simulation stages and its actions, for a multiple run simulation experiment using a time-indexed LFFC

Stage 1:	`Initialize()`
→	Set-up a network connection between 20-sim and Python
→	Set `runNumber` = 0 (run number in simulation)
→	Create folder for storage of simulation data
→	Read parameters from file `parameter.txt`:
	`lowerInput`: lower value of BSN input domain
	`upperInput`: upper value of BSN input domain
	`numberOfKnots`: number of B-splines distributed over BSN input domain
	`splineDegree`: degree of the B-splines
	`evalDegree`: degree of the output evaluation
	`learningRate`: learning rate of the BSN
→	Determine knotvector and internal knots
→	Create initial 3-tuple `(t,c,k)`, with c all zeros
→	Save initial 3-tuple `(t,c,k)`

Stage 2:	`InitializeRun()`
→	Clear arrays in which data is stored
→	Open and store tuple `(t,c,k)` from the previous run
	(or the initialized one, if `runNumber=1`)

Stage 3:	`ProcessStep()`
→	Append data arrays t, u_{FB}, x, v, a and y each sample i
→	Evaluate the output value $u_{FF}[i]$ using `interpolate.splev`
→	Determine target value: $u_{target}[i] = u_{FF}[i](previousrun) + u_{FB}[i] \cdot \gamma$
→	Append data arrays: $u_{target}[i]$ and $u_{FF}[i]$ returned from `interpolate.splev`

Stage 4:	`TerminateRun()`
→	Determine 3-tuple `(t,c,k)` for the next run, using `interpolate.splrep`
	by passing the arrays input, target, the degree of the splines k, and internal knotvector
→	Save to file: 3-tuple `(t,c,k)` to be used in the next run
→	Save arrays to file: time t, feedback u_{FB}, feed-forward u_{FF}, reference x, v, a and output y

Stage 5:	`Terminate()`
→	Close network connection between 20-sim and Python

Table 4.3: B-spline settings, for time-indexed LFFC (* Python only, ** 20-sim only)

Option	**setting**
Input (lower)	0 [s]
Input (upper)	3.78 [s]
Learning rate	0.5 and 0.9 [-]
Spline order	2 (= degree 1) [-]
Evaluation order*	0 [-]
Number of splines	400 and 500 [-]
Learning mode**	learn after leaving a spline
Network type**	discrete

The number of B-splines determine the smoothness and accuracy of the approximation and the learning rate influences the rate at which convergence will occur. For a fixed number of B-splines (400 or 500) the absolute maximum error and the absolute mean error are equal the moment the learning rates are chosen to be 0.5 or 0.9. This is observed for both the 20-sim and the Python implementation.

Furthermore, the larger number of B-splines results in a more accurate approximation and in a lower tracking error for both implementations. A small difference between 20-sim and Python is that the absolute maximum tracking error and the absolute mean error are slightly lower for 20-sim than for Python when "Run 25" is observed using 400 and 500 B-splines.

The tracking errors of the B-spline networks using 400 and 500 splines are shown in Figure 4.10 and Figure 4.11. Together with the tracking errors the reference signals x, v and a are plotted. From observations and from Table 4.4 it is observed that there is no difference in the performances of the simulations the moment the number of B-splines is kept constant (400 or 500) while the learning rates vary (0.5 or 0.9). This can be explained by the fact that the learning rates only influences the rate at which convergence occur and not influence the performance of the system, off course for learning rates that cause no instable behaviour. Since, the simulation results are equal for both learning rates only the results are shown for $\gamma = 0.5$.

Table 4.4: Absolute maximum error and absolute mean error for varying number of B-splines and learning rates

400 B-splines					
		Run 1		**Run 25**	
Error	γ	**Python** [m]	**20-sim** [m]	**Python** [m]	**20-sim** [m]
Abs. maximum	0.5 0.9	$4.500 \cdot 10^{-05}$	$4.500 \cdot 10^{-05}$	$5.144 \cdot 10^{-07}$	$4.821 \cdot 10^{-07}$
Abs. mean	0.5 0.9	$1.122 \cdot 10^{-05}$	$1.122 \cdot 10^{-05}$ [m]	$1.967 \cdot 10^{-08}$	$1.622 \cdot 10^{-08}$

500 B-splines					
		Run 1		**Run 25**	
Error	γ	**Python** [m]	**20-sim** [m]	**Python** [m]	**20-sim** [m]
Abs. maximum	0.5 0.9	$4.500 \cdot 10^{-05}$	$4.500 \cdot 10^{-05}$	$4.218 \cdot 10^{-07}$	$3.480 \cdot 10^{-07}$
Abs. mean	0.5 0.9	$1.122 \cdot 10^{-05}$	$1.122 \cdot 10^{-05}$	$1.874 \cdot 10^{-08}$	$1.152 \cdot 10^{-08}$

It can be seen that the tracking error is increased the moment the reference signal makes a transition from one stage to another (for instance, from no movement (NM) to constant jerk (CY) or from CY to constant acceleration (CA)) . In Figure 4.9 those transition points are marked such that it is more easily to refer to them.

Remarkable to see is that the BSN of Python using 400 splines shows only at 15 transition points an error peak (at transition point (TP) 15 no peak is observed) while the 20-sim implementation shows at every transition point (16) error peaks. Transition point 16 is not a physical transition point and therefor no error is observed at that point. It only indicates the transition from the no move (NM) segment at the end of the forward motion and the NM segment at the beginning of the backward motion. The moment the number of B-splines is increased to 500, both the Python and 20-sim implementation show error peaks at each of the 16 real transition points. For both numbers of B-splines the width of the error peaks is wider and the maximum absolute error is larger for the python implementation.

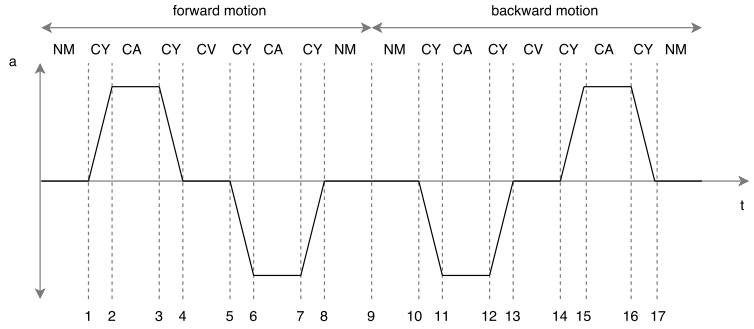
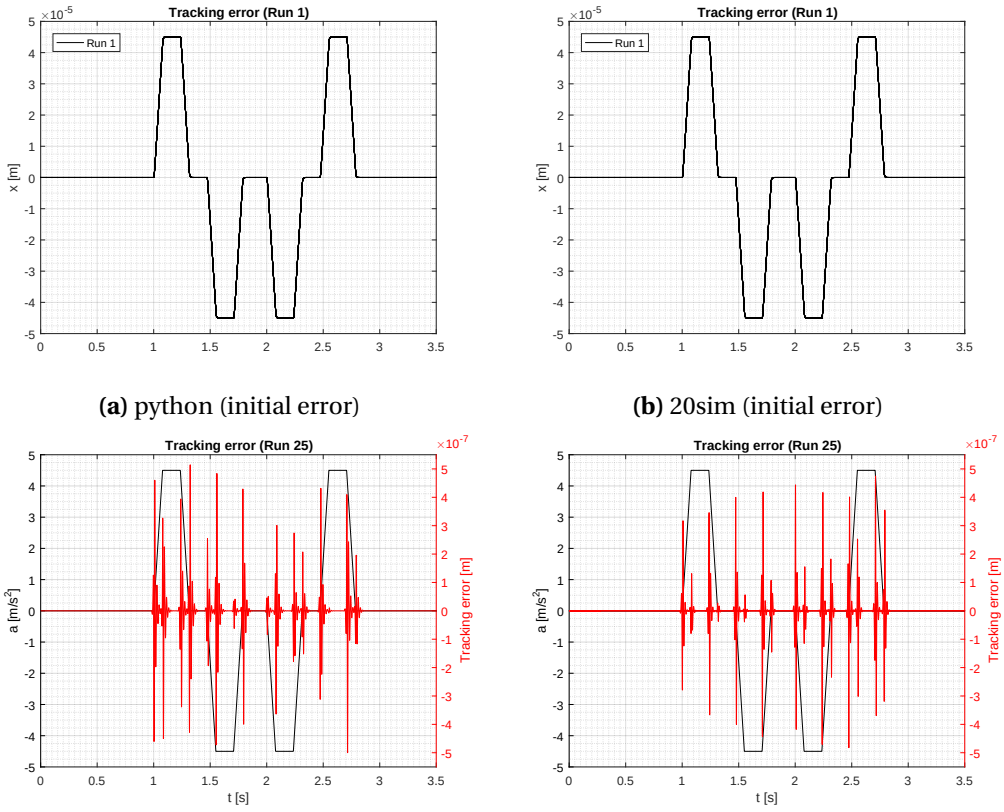


Figure 4.9: Transition points (17) defined at reference acceleration signal. The 18 segments the signal can be divided in are referred to by NM: no move, CY: constant jerk, CA: constant acceleration and CV: constant velocity



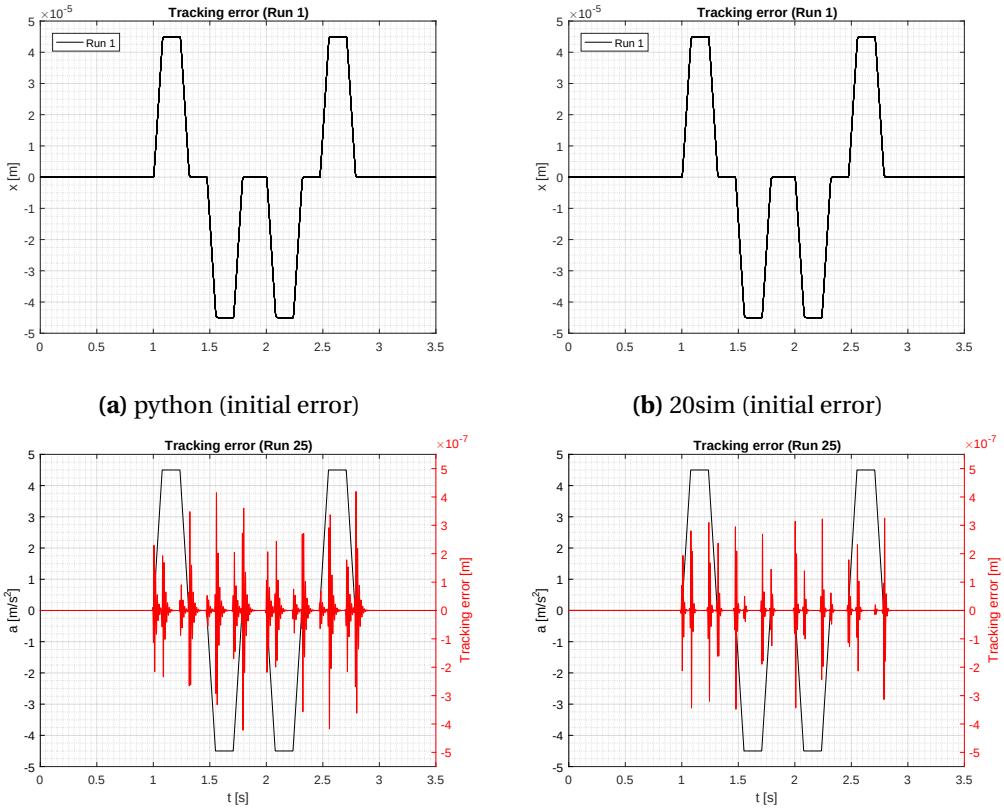
(a) python (initial error)

(b) 20sim (initial error)

(c) python (converged error)

(d) 20sim (converged error)

Figure 4.10: Tracking error (initial = without applying LFFC and converged) using 400 B-splines and $\gamma = 0.5$



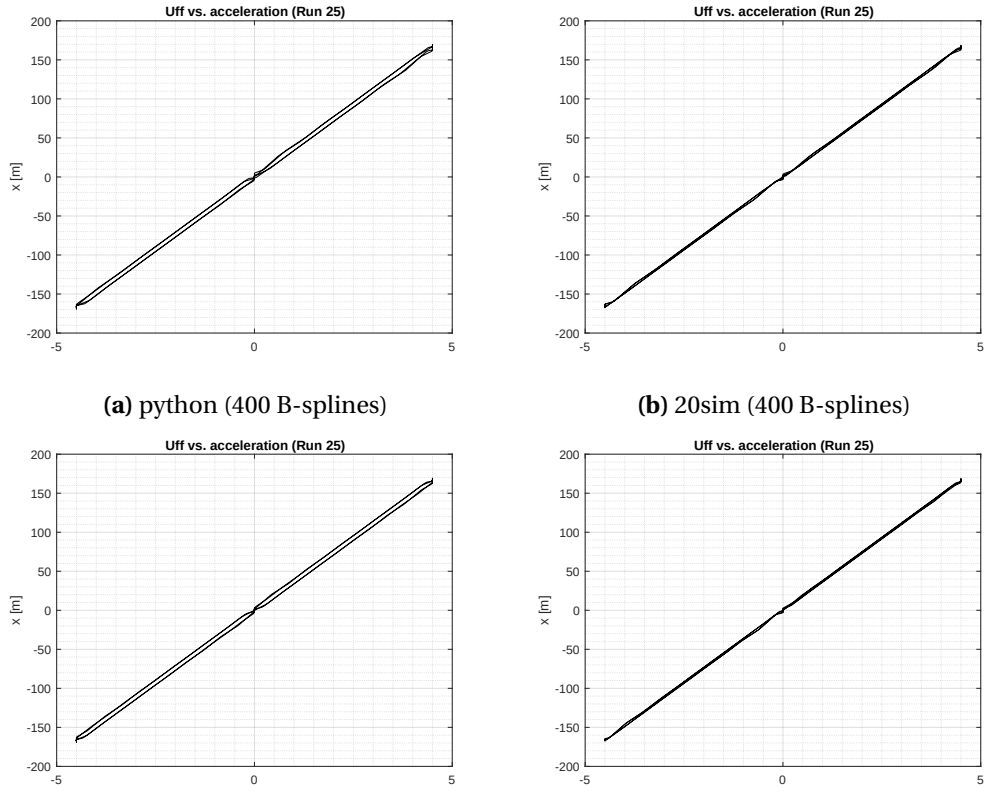
(a) python (initial error)

(b) 20sim (initial error)

(c) python (converged error)

(d) 20sim (converged error)

Figure 4.11: Tracking error (initial = without applying LFFC and converged) using 500 B-splines and $\gamma = 0.5$



(a) python (400 B-splines) **(b)** 20sim (400 B-splines)

(c) python (500 B-splines) **(d)** 20sim (500 B-splines)

Figure 4.12: Feed-forward signal versus acceleration using 400/500 B-splines and $\gamma = 0.5$

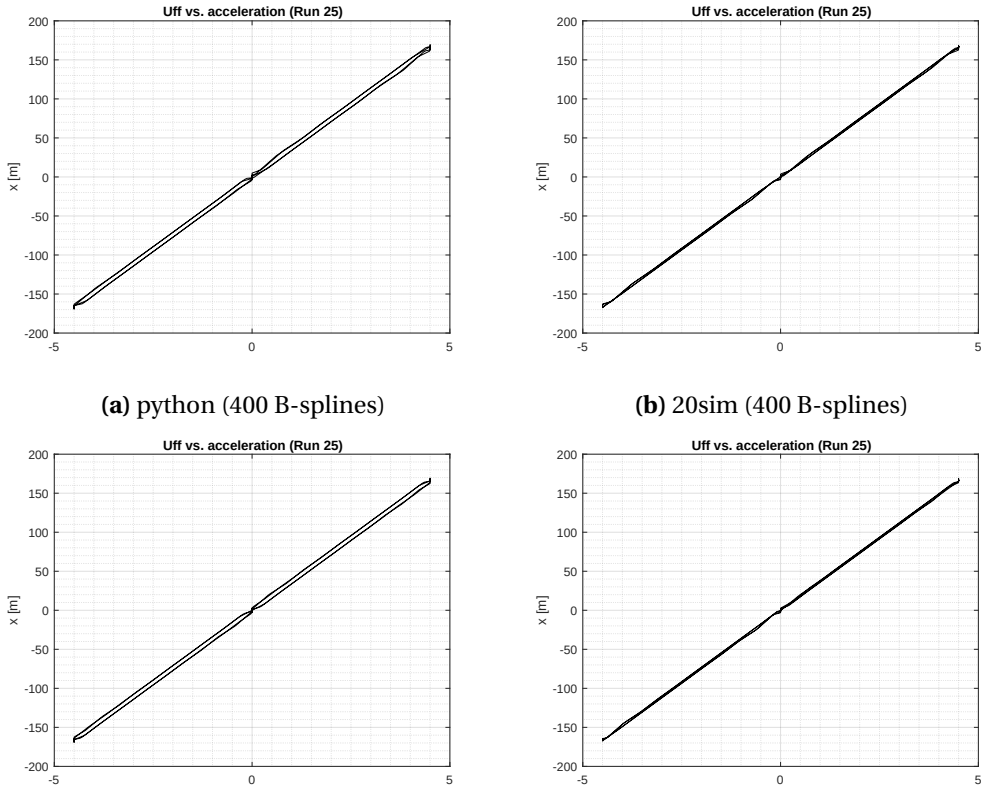
In Figure 4.12 and 4.13 the feed-forward signal u_{FF} is plotted against the reference acceleration for both 400 and 500 B-splines and for learning rates of 0.5 and 0.9. The slope of those lines represents the approximation by the learning controller of mass m_L.

First the plots are observed for $\gamma = 0.5$. The slope of the converged BSN's (Run 25) in the situations using 400 and 500 B-splines are similar for Python and 20-sim. This means that the adaptation of the weights is performed at the same rate. What can be seen as well is that the shape of the learned signal differs.

In observing the plots for $\gamma = 0.9$ similar shape difference are observed between the converged Python and 20-sim implementations. Furthermore, it can be seen that for the larger learning rate both learning controllers converge faster than was observed for $\gamma = 0.5$. This is caused by the fact that a larger learning rate enables the learning controller to adapts its weights faster.

For both implementation of Python (for $u_{FF} vs. a$) similar effect was observed, i.e. the learning controller outputs different values for u_{FF} even though the supplied acceleration is equal. This phenomena is much smaller for the 20-sim implementation and therefor it is checked if there might be a time-delay in the Python implementation.

The difference between the feed-forward signals of 20-sim and Python of Run 25 are plotted against the time, see Figure 4.14. The reference acceleration is shown in the same graph. The expectation of the appearance of a time-delay can be confirmed by the graph. A difference of about 2 [N] is observed in the sections of constant jerk. The segments that can be described by a constant acceleration, constant velocity or no movement do not show such a difference and the fluctuations are observed around zero.



(a) python (400 B-splines)

(b) 20sim (400 B-splines)

(c) python (500 B-splines)

(d) 20sim (500 B-splines)

Figure 4.13: Feed-forward signal versus acceleration using 400/500 B-splines and $\gamma = 0.9$

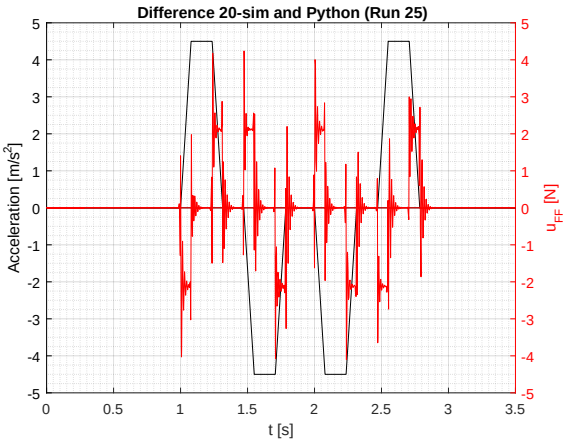


Figure 4.14: Difference between u_{FF} of 20-sim and Python plotted against the time for Run 25.

In Figure 4.15 the same graph is shown for which now the data from 20-sim is shifted one sample time to the left. The differences between both implementations now shows all fluctuations at the transition points around zero. The "hysteresis" like behavior observed in the graphs u_{FF} vs. a indeed were caused by a time delay.

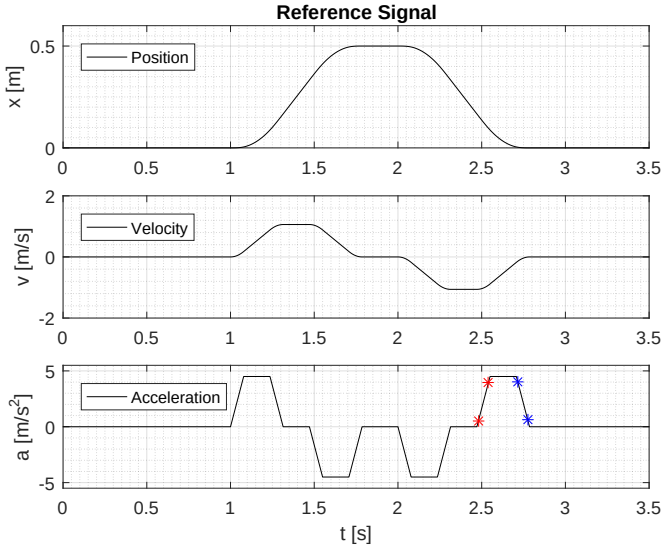


Figure 4.16: Reference motion with markers indication the ranges over which the value of the approximated mass is determined, left) accelerating segment and right) decelerating segment

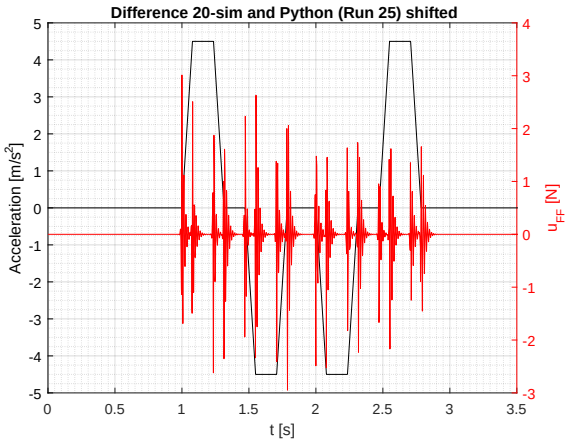


Figure 4.15: Difference between u_{FF} of 20-sim and Python plotted against the time for Run 25, for which 20-sim is shifted one time sample to the left.

It might be possible that the implementation in 20-sim sends its data towards 20-sim the moment the B-spline editor has done its job, one sample time later than the Python implementation, further research needs to be done to verify the proposed cause.

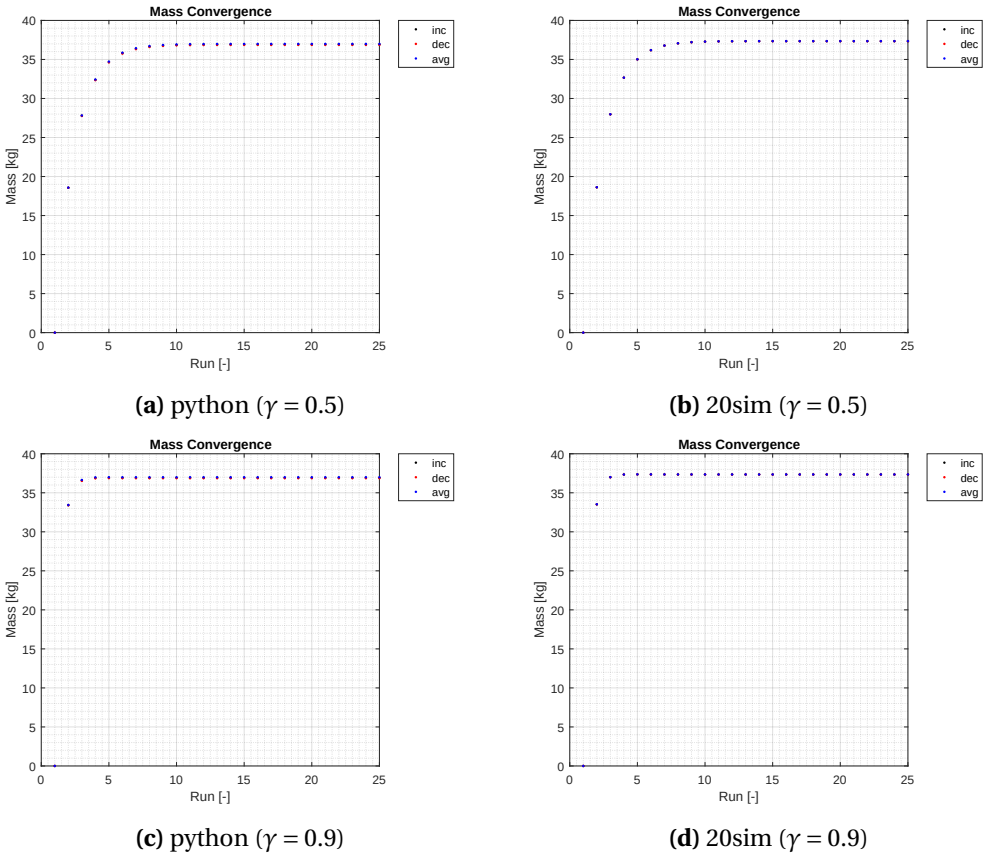
The value the learning controllers approximate can be found by calculating $\Delta U_{FF}/\Delta a$. This takes place at the last part of the reference motion at both the increasing (left markers) and decreasing (right markers) part of the acceleration, see Figure 4.16. Besides that, the averaged value over both parts is determined, see Table 4.5.

It can be seen that the moment 400 B-splines are used both Python and 20-sim approximate a slightly lower approximation at the increasing part than at the decreasing part. The moment 500 B-splines are chosen the results are reversed, i.e. the decreasing part shows slightly lower approximation than the increasing part. On average Python deviates -0.071 [kg] and +0.057 [kg] from m_L for respectively 400 and 500 B-splines. The implementation of 20-sim shows on average a deviation of +0.344 [kg] and -0.104 [kg] from m_L using 400 and 500 B-splines. As a result, the average deviation of Python is smaller in all cases considered.

Table 4.5: Mass approximation for various number of B-splines and learning rates

		400 B-splines		500 B-splines	
		Python	**20-sim**	**Python**	**20-sim**
Increasing	0.5 0.9	36.866 [kg]	37.337 [kg]	37.068 [kg]	36.785 [kg]
Decreasing	0.5 0.9	36.991 [kg]	37.350 [kg]	37.046 [kg]	37.007 [kg]
Average	0.5 0.9	36.929 [kg]	37.344 [kg]	37.057 [kg]	36.896 [kg]

The rate at which the learning controllers converge can be nicely seen by plotting the mass approximation against the run number, see Figure 4.17. Since, the number of B-splines used does not influence the rate of convergence only the results are shown using 400 B-splines. As expected, the approximations for $\gamma = 0.9$ reach convergence in less runs than the moment $\gamma = 0.5$. No significant difference can be observed in the rate at which convergence occurs when comparing 20-sim and Python.



(a) python ($\gamma = 0.5$)

(b) 20sim ($\gamma = 0.5$)

(c) python ($\gamma = 0.9$)

(d) 20sim ($\gamma = 0.9$)

Figure 4.17: Rate of convergence using BSN's using 400 B-splines and learning rates of 0.5 and 0.9

4.1.3.2 Implementability

Separating both controller parts creates more freedom in the use of applications at both sides. It makes it possible to implement the feed-forward part in Python and to use it in combination with a plant model in 20-sim.

As was described in Section 2.4 a linear motor model is used as the illustrative example. In stead of the control of a 20-sim model the network communication enables the option (after

proper system adaptations) to control a real physical linear motor. One of the adaptations that has to be done is transforming the Python implementation (part of) into something useful for real-time. For that, it is only necessary to transform the evaluation part of the approximator. Python itself is not developed for real-time implementations. The control of a real physical linear motor is beyond the scope of this thesis and so is the implementation of the feed-forward useful for real-time use.

The Python implementations gained a lot of freedom in the design of the B-spline network. But not only for the B-spline Network, it is now even possible to use other types of function approximators as long as they can be implemented in Python. The parameters of the BSN are more easily to set and more of them can be changed. For instance:

- → define a non-uniform knotvector
- → define a knotvector with multiplicity of knots
- → BSN output evaluations of derivatives
- → BSN output evaluation of one point or complete array

Besides the parameter settings it is more easy to perform a multiple run simulation experiment. Python enables to automatically save data to file after a run and to load parameters `(t,c,k)` the moment a new run starts. Within 20-sim the network weights needed to be stored and loaded by hand, which is very impractical for multiple run experiments.

Thanks to the network connection, the simulation results can be easily stored and loaded into an evaluation program, like Matlab. This enables fast reproduction of the simulation results.

4.1.4 Conclusion

The comparison between 20-sim and Python is done based on two points of view. One point of view evaluates the performance of the learning controllers and the other evaluates the user-friendliness and the freedom the user has in order to perform the function approximation.

From the simulations the performances of the learning controllers is obtained. For a fixed number of B-splines (400 or 500) both implementations show almost identical convergence values. The Python and 20-sim learning controllers can therefore be said to operate consistently and their outcomes are independent of the learning rate.

The rate at which both learning controllers converge is similar for both learning rates (0.5 and 0.9). Although, the approximations of m_L deviates little bit more for 20-sim than for Python and the average approximation of Python is closer to $m_L = 37$ [kg] in either case. The larger deviation in Python is observed in the plot u_{FF} versus a in the form of hysteresis (or in fact a time-delay). This phenomena is lower for 20-sim than for Python. The time-delay is caused by a delay observed between both implementations, Python is one step (1 sample instance) ahead of the 20-sim.

In all situations considered the absolute maximum tracking error and the absolute mean error is lower for 20-sim than for Python. From this can be concluded that the performance of 20-sim is slightly better than Python. Although, the tracking errors of 20-sim are slightly lower, the BSN implementations of Python and 20-sim can both be used to perform the function approximation in the feed-forward part of the control system.

The comparison between 20-sim and Python based on the performance is competitive, but the moment both are compared based on the user-friendliness and the user's freedom Python definitely scores better. By setting up a BSN in Python the user has a lot of freedom in selecting and defining the network parameters. The parameters are more easily to set and more parameters can be selected.

Besides that, it is more easily to set-up an automated multiple run simulation experiment in which data is automatically supplied and stored to file. This must be done by hand in 20-sim.

4.2 State-Indexed LFFC

The moment non-repetitive motions must be performed a time-indexed learning controller will not function at its best. Each time a new motion is supplied the learning process has to start all over again and its previously gained knowledge is lost. To overcome this problem a state-indexed learning controller can be used. The periodic motion time (T_P) is no longer used as the input but the reference signal (and/or derivatives thereof). In this section a 1-Dimensional state indexed LFFC is designed that has to approximate the mass m_L of the plant and therefor the reference acceleration is supplied to the learning controller. The structure of the time-indexed LFFC is shown in Figure 4.18.

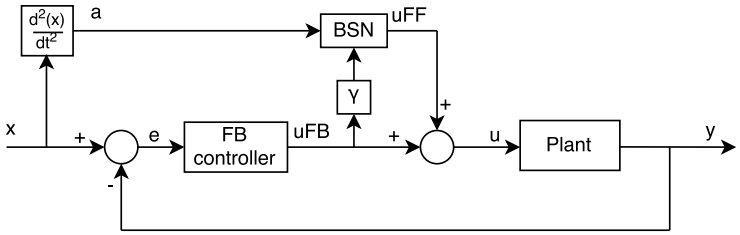


Figure 4.18: State-indexed LFFC structure (1-dimensional)

4.2.1 Design

The learning controller has to learn the inverse behavior of the plant. The same ideal plant model is used as was used in the case of the time-indexed LFFC. The model includes the inertia of the mass only, see Figure 4.19.

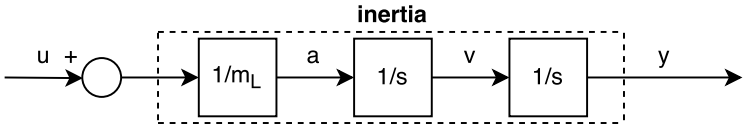


Figure 4.19: Ideal plant model used for the 1-dimensional state-indexed LFFC

The design of a state-indexed LFFC is performed along the same step-by-step plan as was used for the time-indexed LFFC. The parameters to be selected are the upper and lower input values (accelerations), the learning rate γ, the number of B-splines and its distribution and the degree of the splines.

Step 1: Input selection of the BSN
In order to learn the inverse behavior of the plant (mass m_L) an appropriate choice is to select the second derivative of the reference position, i.e. the acceleration a [m/s^2]. This can be traced back by observing Newtons second law: $F = ma$.

Step 2: Selection of the B-spline order
Similar to the time-indexed LFFC first degree (second order) B-splines are used for the B-spline network. This type of B-splines enables smooth enough function approximation while limiting computational process time.

Step 3: Selection of the trainings motion
The type of trainings motion used for the state-indexed LFFC is a partial cubic motion profile (see Appendix A). Instead of supplying the reference motion once during a simulation session the motion is repeated five times. As a result, the learning controller is supplied with target values that correspond to the same acceleration multiple times. By taking proper care of the duplicates the rate at which the learning controller convergence can be increased. The refer-

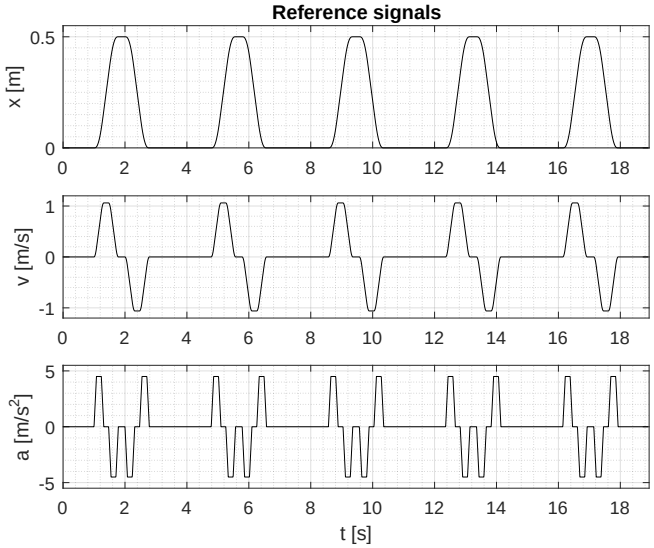


Figure 4.20: Reference signal used as trainings motion for the 1-dimensional state-indexed LFFC

ence motion is limited to achieve an acceleration of 4.5 [m/s^2] over a range of motion of 0.5 [m]. The signal is shown in Figure 4.20 and its parameters in Table 4.6.

Step 4: Selection of the B-spline distribution and the number of splines
From Newton's second law it is known that there exists a linear relation between the force exerted on a mass and its acceleration. The number of B-splines chosen for the B-spline Network specifies the amount of freedom of the function approximator. The larger the number of B-splines the more freedom is gained as there are more points the approximator will fit its data on. Learning a linear relation requires two B-splines only, though an extra B-spline is added in zero. The third B-spline tries to force the function approximator to achieve for zero reference acceleration a zero the feed-forward output (i.e. the force) as well. The two other B-splines are placed at the lower an upper acceleration inputs, see Figure 4.21

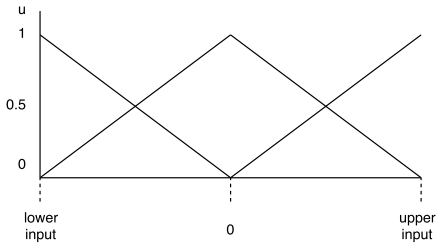


Figure 4.21: B-spline distribution feed-forward controller compensating for the inertia of the mass

Step 5: Selection of the learning rate
The learning rule introduced for the time-indexed BSN can no longer be used as the motion period for signals supplied to a state-indexed LFFC may fluctuate. A mediate learning rate of $\gamma = 0.5$ is selected for training the state-indexed learning controller.

4.2.2 Implementation

The learning controller of the 1-dimensional state-indexed LFFC is implemented using the Python Scikit-learn library. This controller is an extension of the general control system that consists of a feedback controller and an ideal plant model, see Figure 4.22.

Table 4.6: Trainings signal parameters state-indexed LFFC, for $i \in [1,5]$ the repetition number of the partial cubic wave

Parameter	Value	Unit	Description
T_{rise}	$0.786 + i * T_P$	s	Rise time
T_{start}	$1.000 + i * T_P$	s	Start time
T_{stop}	$1.786 + i * T_P$	s	Stop time
T_{return}	$2.000 + i * T_P$	s	Return time
T_{end}	$2.786 + i * T_P$	s	End time
T_p	3.786	s	Period of signal
T_{total}	18.93	s	Total motion time (five partial cubic waves)
j_{max}	57.276	m/s^3	Maximum jerk
a_{max}	4.500	m/s^2	Maximum acceleration
v_{max}	1.061	m/s	Maximum velocity
x_{max}	0.500	m	Maximum displacement (= stroke)
CV	20	%	Percentage Constant Velocity (CV)
CA	20	%	Percentage Constant Acceleration (CA)

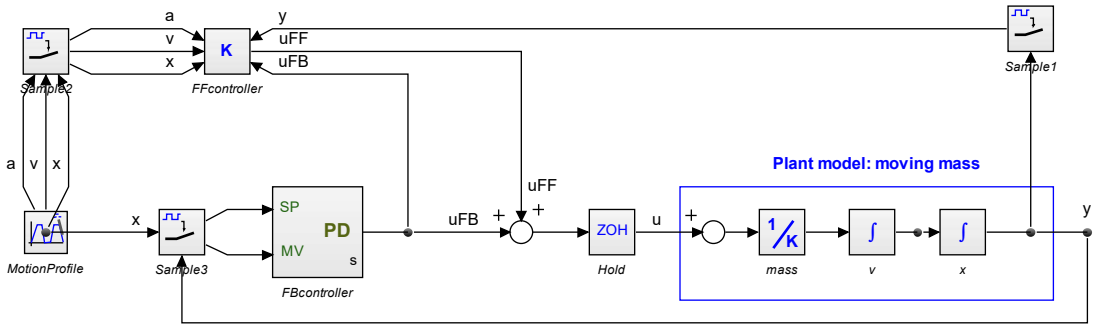


Figure 4.22: 20-sim model for state-indexed LFFC using an ideal plant

The model contains the block "FFcontroller" which is a wrapping around the feed-forward part of the control system. This block captures data but also performs the function approximation of the learning controller. The block has five inputs, i.e. reference position x [m], reference velocity $v = \dot{x}$ [m/s], reference acceleration $a = \ddot{x}$ [m/s^2], feedback output u_{FB} [N] and the system output y [m]). The output of the block is the feed-forward signal u_{FF} [N].

Each sample time (1 [ms]) the data is captured and sent over the network to Python. At the Python side the data is manipulated (performing the function approximation) and its output is send back to 20-sim, i.e. the output of feed-forward part u_{FF} [N]. Take a look at Chapter 3 for more information about the network connection and the communication between 20-sim and Python.

The function approximator is implemented as a B-spline network by making use of the Python Scikit-learn library by using the functions `interpolate.splev` and `interpolate.splrep`. The first function evaluates the output of the learning controller and the second returns a 3-tuple `(t,c,k)` containing the knotvector `t`, the coefficients (network weights) `c` and the degree of the B-splines `k`.

The input data supplied to `interpolate.splrep` must be unique, sorted in ascending order and must be matched to the knotvector. In the case of a time-indexed LFFC those requirements were met automatically as long as the sample time was (much) smaller than the time

difference between two consecutive knots. That those requirements are met for a state-indexed LFFC is not necessarily the case. Therefor some data preprocessing is applied to the input data:

1. **Make the input data unique**
   By observing the input-target mapping $a[i], u_{target}[i]$, in which i represents a certain sample instance, several methods can be applied in order to make the data set unique.

   The methods are explained by first assuming an input-target mapping in which the input data has multiplicity of n. The input dataset can be described by $\mathbf{x} = [x_1, x_2, ..., x_n]$ with $x_1 = x_2 = ... = x_n$ and the target data by $\mathbf{y} = [y_1, y_2, ..., y_n]$.

   In Table 4.7 six methods are presented in order to make the input-target mapping unique:

   Table 4.7: Methods considered to remove duplicated values of the input-target mapping

Method	Description
Minimum	Use the minimum value in **y**
Maximum	Use the maximum value in **y**
First	Use the first value in **y**
Last	Use the last value in **y**
Mean	Take the mean value of all elements in **y**
Random	Pick a random value out of all elements in **y**

   Based on simulation results one method is selected that will be used throughout the rest of the simulations for the state-indexed LFFC in this thesis. Multiple run simulation experiments are performed for which each consists of 15 runs, using 3 B-splines and a learning rate of 0.5. The 20-sim model used for this is shown in Figure 4.22 and the reference input signal is the partial cubic motion created by the waveform generator of 20-sim (maximum acceleration of 4.5 and a maximum range of motion (back and forth) of 0.5 [m]), see Figure 4.16.

   In Figure 4.23 the tracking error of Run 15 is shown and in Figure 4.24 a zoomed-in version at the first maxima, the first minima and the part in between the back and forward motion.

   By observing Run 15 the lowest error is observed for "maximum" and "first" at the maxima and at the minima the best error is shown by "minimum". In the latter case the worst error is observed for "first" and from this can be concluded that there exists no specific relation between the best performing methods at the maxima, "maximum" and "first" and at the minima, "mininum" and "first" (at minima).

   The "average" and "random" methods score roughly the same at the minima and maxima peaks. At the parts in which no movement is performed the "average" method shows an error that is more close to zero than the "random" method, which is wishful. The closer to zero the tracking error the better the linear motor will stay located at its non-moving position. It is then less likely to occur that the linear motor will deviate from its original start position after couple of runs.

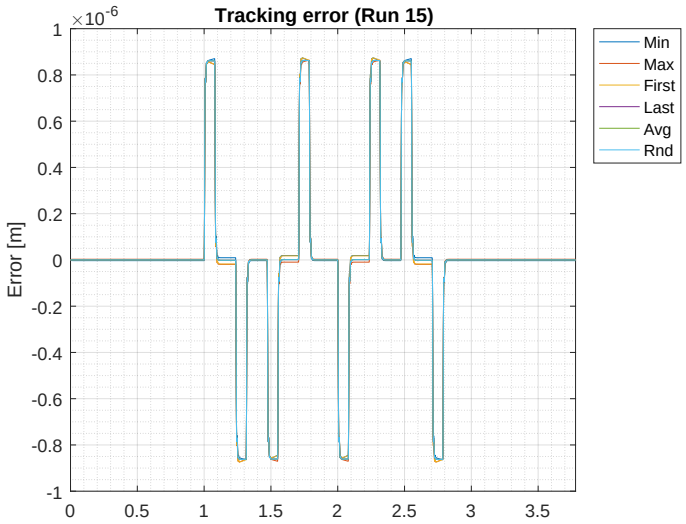


Figure 4.23: Tracking error using various methods in order to remove duplicates (Run 15)

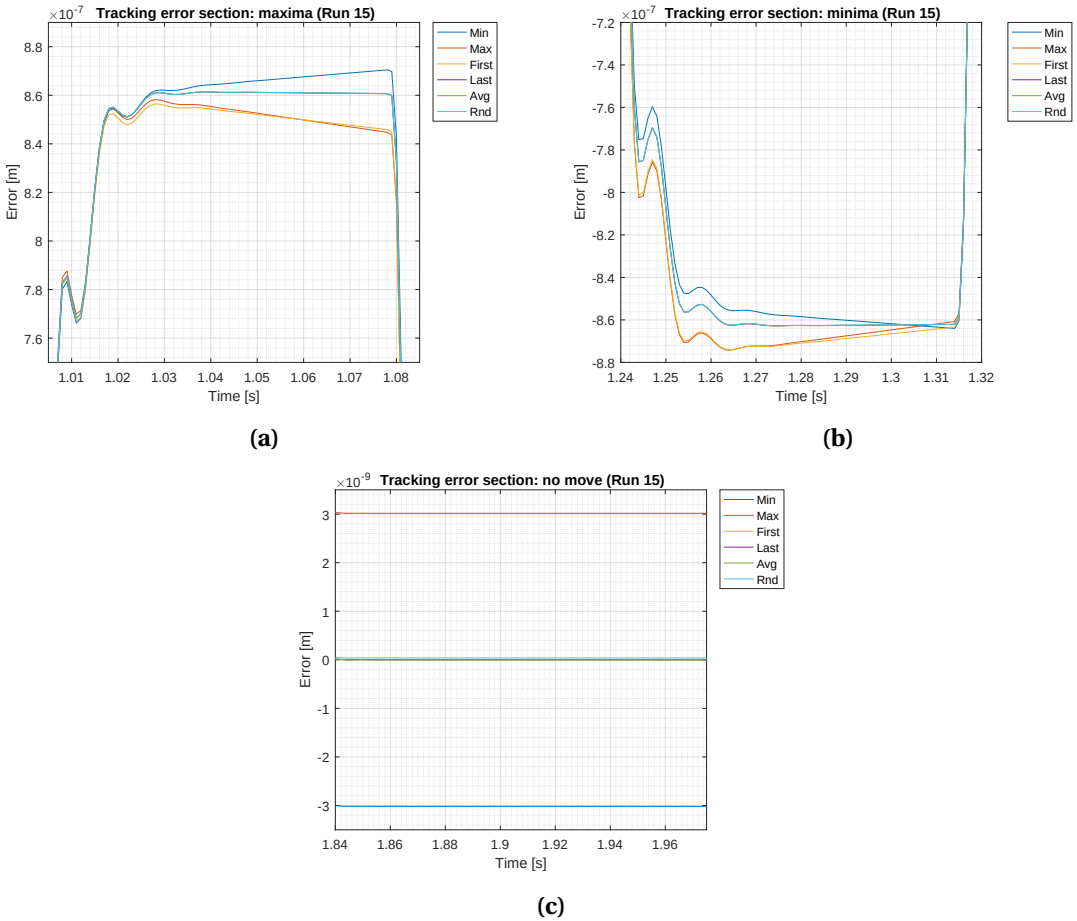


Figure 4.24: Tracking error zoomed-in at a) maxima, b) minima and c) non-moving part in between the forward and backward motion. Observed for various methods in order to remove duplicates (Run 15)

In Table 4.8 the maximum absolute tracking errors and the absolute mean tracking errors are listed for Run 15, for all six methods. The lowest errors appear using the methods "last", "average" and "random".

Table 4.8: Maximum absolute tracking error and absolute mean tracking error using various methods in removing duplicates out of the input-target mapping (Run 15)

	Abs. mean tracking error [m]	**Max. abs. tracking error [m]**
Minimum	$1.473 \cdot 10^{-07}$	$8.743 \cdot 10^{-07}$
Maximum	$1.473 \cdot 10^{-07}$	$8.743 \cdot 10^{-07}$
First	$1.457 \cdot 10^{-07}$	$8.741 \cdot 10^{-07}$
Last	$1.435 \cdot 10^{-07}$	$8.628 \cdot 10^{-07}$
Average	$1.435 \cdot 10^{-07}$	$8.628 \cdot 10^{-07}$
Random	$1.435 \cdot 10^{-07}$	$8.628 \cdot 10^{-07}$

Besides the tracking error the learned signal is observed and the value to which the learning controller converges. In Figure 4.25 the feed-forward signal is plotted against the acceleration for Run 15 and in Figure 4.26 the approximation of the mass (slope of u_{FF} vs. a) per run is shown (see Table 4.9 for the values).

At first sight no significant difference are observed in using various methods in removing duplicated input data. A closer look at the plot u_{FF} vs. a shows that for the methods "random" and "average" their lines pass closer through zero than the other methods. This means that those two for $a = 0$ [m/s^2] a feed-forward signal is produced having a close to zero value as well.

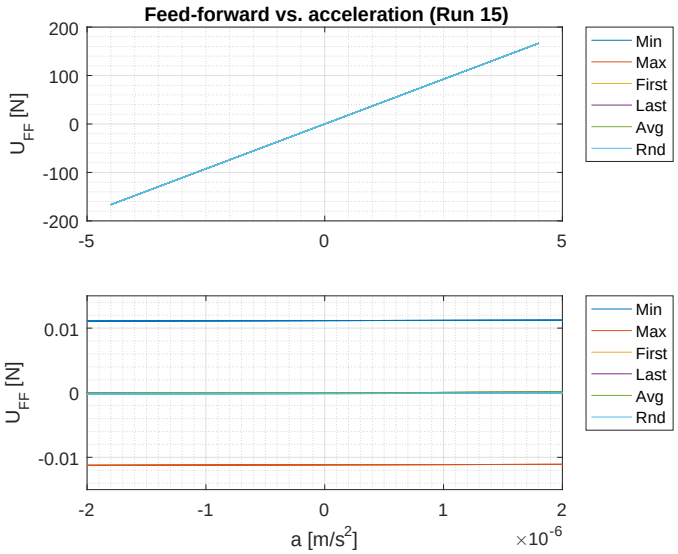


Figure 4.25: Feed-forward signal against acceleration using different methods in removing duplicates out of the input-target mapping for run 15. Top figure shows complete signal and bottom signal is a zoomed-in version around zero.

From the mass approximation (4.26) can be concluded that again the "average" and "random" value show best performance. Although the differences are small, the value to which the learning controller converges by using those two methods is closest to the mass defined in the plant model $m_L = 37$ [kg]. Table 4.9 shows the approximated mas per run and no significant differences can be observed. The only fact that can be seen is that the methods "maximum" and "first" deviate little bit more than the other methods.

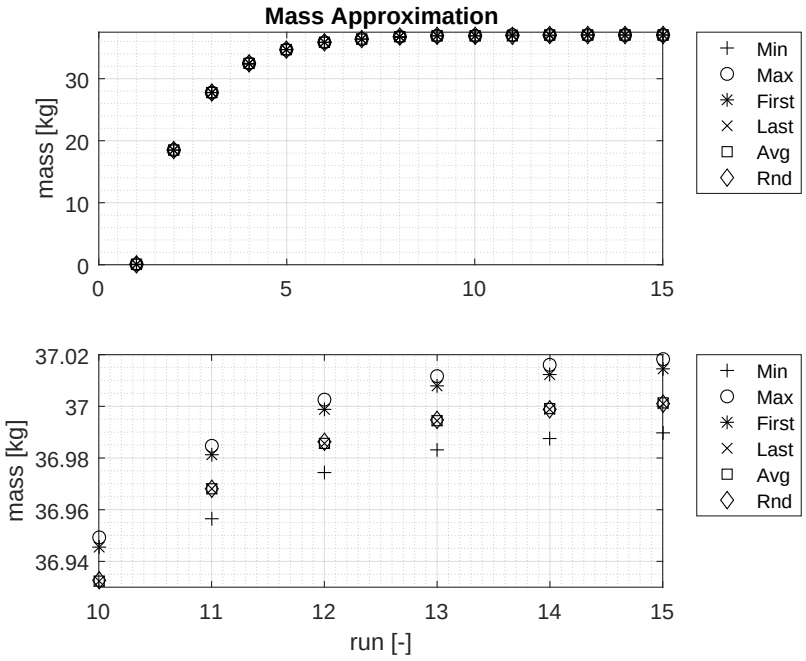


Figure 4.26: Approximated mass per run for various methods in removing duplicates

Table 4.9: Mass approximations of several runs using different methods in removing duplicates out of the input-target mapping

	Mass approximation per method [kg]					
Run	**Minimum**	**Maximum**	**First**	**Last**	**Average**	**Random**
1	0	0	0	0	0	0
2	18.539	18.548	18.542	18.539	18.539	18.539
3	27.790	27.802	27.796	27.790	27.790	27.790
4	32.406	32.419	32.415	32.406	32.406	32.406
5	34.704	34.724	34.720	34.406	34.709	34.707
⋮						
15	37.001	37.018	37.014	37.001	37.001	37.001

To conclude, in observations from the learned signal and the approximated masses is found that both "average" and "random" perform best. From the tracking error the "average" method is recommend even though it does not show lowest tracking errors at the minima and maxima. It does show the closest to zero tracking error at the no movement segments and therefor this method will be used from now on.

2. **Sorted input data (ascending order)**
   The input-target mapping is "zipped" together before the function `sorted` is used to sort both in ascending order. The additional "zipping" action is performed to make sure that during the sorting process the indices's of both arrays remain paired.

3. **Input matched to knotvector definition**
   The moment a motion pattern is supplied that does not spread out its input-target data along the whole range of the knotvector the function `interpolate.splrep` throws an error. In between two consecutive knots an input-target set must be available such that the Schoenberg-Whitney condition is satisfied.

Assume for instance the knotvector $t = [-2,-1,0,1,2]$ and an obtained input-target set $a, u_{target} = [-1.2,-0.9,1.6]$. An error is thrown due to the lack of a data point in between $t = 0$ and $t = 1$. To overcome this error the input-target mapping will be manipulated, referring to this as the method "Manipulate input-target mapping". In order to explain this method a multiple run simulation experiment is assumed.

Manipulate input-target mapping
This method adapts the input-target mapping by adding data points. Before Run 1 starts an input-target mapping is created such that in between two consecutive knots (halfway) a data point is created according to:
$a[i] = \frac{t[i+1]-t[i]}{2}$
u_{target} = output of `interpolate.splev`

For data captured in Run 2 and further, data that is collected within a section of two consecutive knots will overwrite the data previously there (the initially created data or data obtained in a previous run). The same is done for data located at exactly a knot point, the previous data will be overwritten by the current data. This is illustrated in Figure 4.27 for Run 0 (initializing data set, assumed no previous data present such that each added data points has zero target value), Run 1 and Run 2.

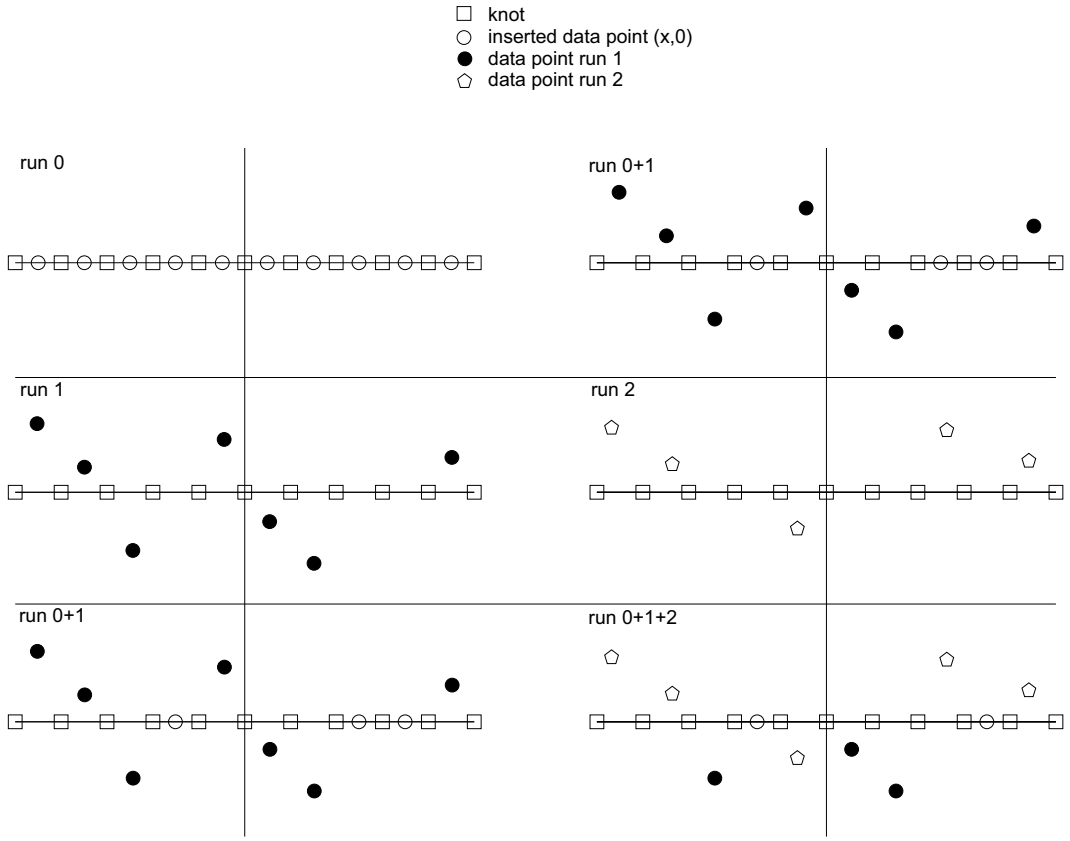


Figure 4.27: Manipulate input-target mapping to match to the defined knotvector

Besides the requirements from `splrep` some additional data processing is performed. The value of the target data is compared to the lower input and upper input parameters set for the B-spline network. A data point (a, u_{target}) for which the $u_{target} < lowerInput$ or $u_{target} > upperInput$ are removed out of the data set, as those will never have a positive influence on the function approximation. This is carried out by the function `matchInputToParameters()`.

Compared to the time-indexed Python implementation some functionality is added, such as that the user is now able to choose what task the learning controller has to perform. For more detail about the implementation (five stage communication structure and list of executed functions per task) see Appendix C.

4.2.3 Simulations

The abilities of the Python implementation of the B-spline network for a 1-dimensional state-indexed LFFC is performed in this section. A multiple run simulation experiment (25 runs) is set-up using the model shown in Figure 4.22. The learning controller will start learning from zero initial knowledge (`Task = 1`) and will try to learn mass m_L of the plant.

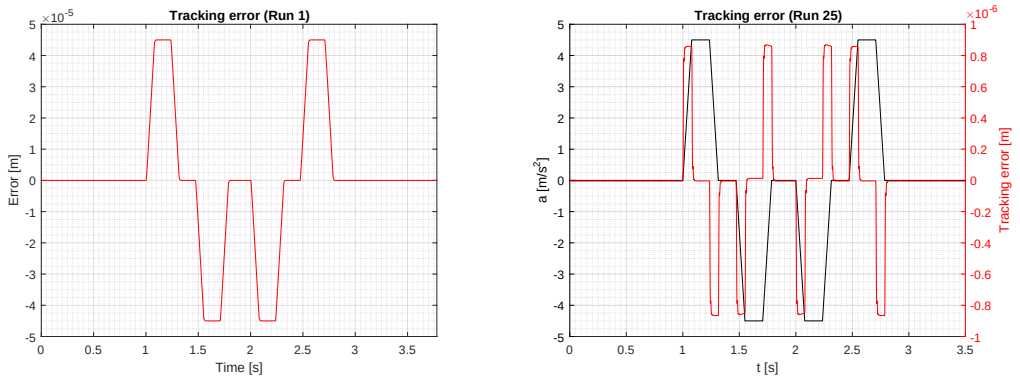
The performance is checked by observing the tracking error, the learned signal and the convergence of the learning controller. The reference signal used for this is the partial cubic motion having a maximum acceleration of 4.5 [m/s^2] and has a for- and backward range of motion of 0.5 [m], see Figure 4.20.

A summary of the B-spline network parameter is given in Table 4.10. In the previous section the method chosen to remove duplicates was "average" and the method "Manipulate input data" is used to match the input to the knotvector.

Table 4.10: B-spline parameter settings of Python implementation for state-indexed LFFC

Option	**setting**
Input (lower)	-5 [m/s^2]
Input (upper)	5 [m/s^2]
Learning rate	0.5 [-]
Spline order	2 (= degree 1) [-]
Evaluation order	0 [-]
Number of splines	3 [-]

The absolute maximum and mean tracking error are shown in Figure 4.28, together with the reference signals x, v and a for Run 1 (no learning applied) and Run 25. The absolute maximum value has dropped by a factor 50 from $4.500 \cdot 10^{-05}$ towards $8.69 \cdot 10^{-07}$. The absolute means has dropped by a factor 75, see Table 4.11. All the errors occur at the transition points and in between those sections the tracking error is within the range of $\cdot 10^{-09}$. The plots do not show any strange behavior.



(a) Initial **(b)** Converged

Figure 4.28: Tracking error using 3 B-splines and a learning rate of 0.5 for a 1-dimensional state-indexed LFFC

Table 4.11: Absolute mean and maximum error observed using a 1-dimensional state-indexed LFFC

	Tracking error	
Run	**Absolute maximum** [m]	**Absolute mean** [m]
1	$4.500 \cdot 10^{-05}$	$1.121 \cdot 10^{-05}$
25	$8.695 \cdot 10^{-07}$	$1.466 \cdot 10^{-07}$

The approximation of the mass, per run is shown in Figure 4.29. The value the learning converges to is 37 [kg], see Table 4.12. The learning rate was set to 0.5 and within 10 runs the BSN approximates a value that only deviates 0.07 [kg] from Run 25.

In the time-indexed LFFC a fluctuation was seen in the determination of the slope at the increasing and decreasing part of the reference signal (see the marks, shown in Figure 4.16). No fluctuation is observed in the case of the state-indexed LFFC.

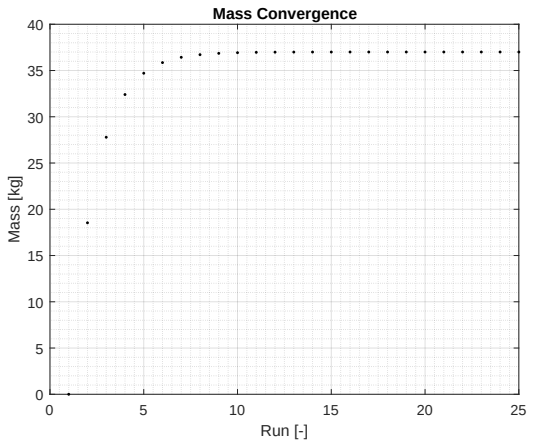
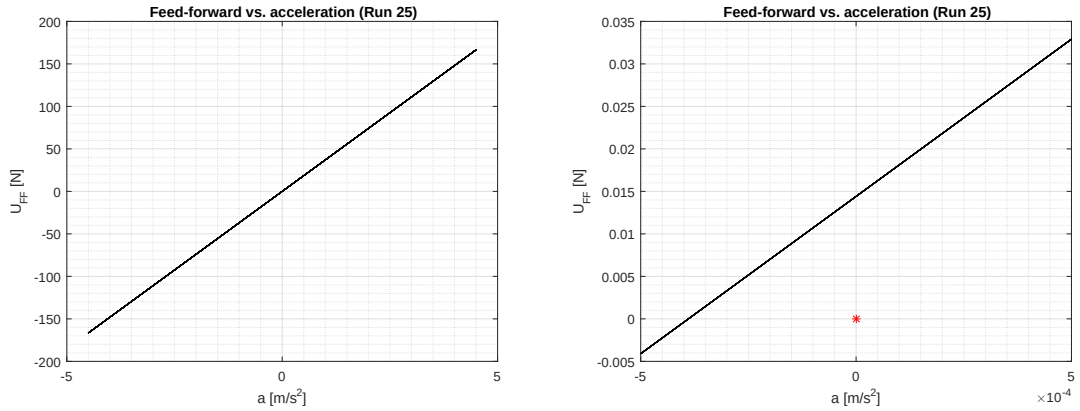


Figure 4.29: Mass approximation per run of the 1-dimensional state-indexed learning controller using a learning rate of 0.5

Table 4.12: Mass approximation after 25 runs by the 1-dimensional state-indexed LFFC

Mass approximation (Run 25)	
Increasing Decreasing Average	37.000 [kg]

The feed-forward vs. acceleration is shown in Figure 4.30 for Run 25, the slope represents the approximated mass by the learning controller. As the time-indexed LFFC showed a time-delay this cannot be observed for the state-indexed LFFC. This means that for the same acceleration value supplied to the BSN the approximator evaluates equal output and returns the same u_{FF} signal.



(a) **(b)**

Figure 4.30: Feed-forward signal plotted against acceleration a) complete input range and b) zoomed in around zero ("*" marks origin)

By zooming in around zero in the plot of Run 25, it is observed that the approximation does not pass exactly through zero. Although, its deviation is small: for $a = 0$ [m/s^2] $u_{FF} = 0.014$ [N], see Figure 4.30b).

4.2.4 Conclusion

The state-indexed LFFC showed improved performance the moment the learning controller was converged. Within the first 10 runs the controller can be said to converge as the difference between Run 25 and Run 10 is only 0.07 [kg]. Compared to a control system without an LFFC the absolute maximum tracking error has dropped by a factor 50. Furthermore is the learning controller able to output the same u_{FF} signal the moment an acceleration is supplied to the BSN multiple times (this was not the case in the time-indexed LFFC).

5 Two Dimensional LFFC

The control systems described in Chapter 4 all used ideal linear motor models. Unfortunately, mechatronic motion systems in the real world are influence by more than one factor. In this chapter a 2-dimensional (2D) state-indexed LFFC is designed and implemented using the Python Scikit-learn library. The learning controller consists of a B-spline network as its function approximator and its tasks is to learn the behaviour of the plant caused by the inertia of the mass and the position dependent cogging.

The implementation of a 2-dimensional LFFC can be done by the so called parsimonious LFFC, T.J.A.de Vries et al. (2001), de Kruif and de Vries (2000) (one-dimensional BSN) or by using multidimensional learning controllers (two-dimensional BSN). The difference between both is found in the way the training of the individual learning controllers takes place. The one-dimensional method learns the influencing factors independent of each other while the two-dimensional method tries to learn both factors at the same time.

Along with the two-dimensional method comes the curse of dimensionality, which is not beneficial. The two-dimensional BSN is only treated by describing its working principle and what is needed to implement it in Python. More research has to be done on this method and especially in how it operates and how it performs during simulations.

5.1 Parsimonious LFFC

In this section the 2-dimensional state-indexed LFFC is designed and implemented according to the structure shown in Figure 5.1. The feed-forward part consists of two 1-dimensional BSNs, supplied to each BSN is one input.

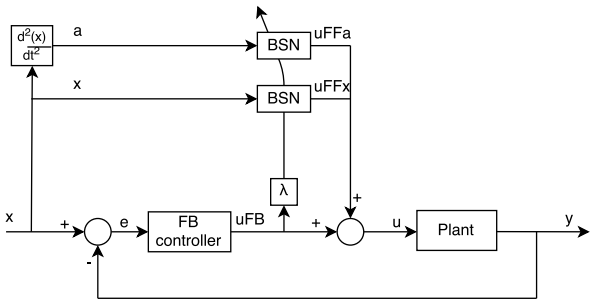


Figure 5.1: Structure of a 2-dimensional state-indexed LFFC (2x 1D-BSN)

Each input is supplied to its own BSN and will returns their own output. Both outputs are added up and fed into the control system just before the plant input:

$$u_{FF} = u_{FFx} + u_{FFa} \tag{5.1}$$

with,

u_{FF}	total feed-forward signal [N]
u_{FFx}	feed-forward signal according to position BSN [N]
u_{FFa}	feed-forward signal according to acceleration BSN [N]

The task that each BSN has to learn is derived from the plant model used, see Figure 5.2. The phenomena included in the model are the inertia of the mass and the position dependent cogging.

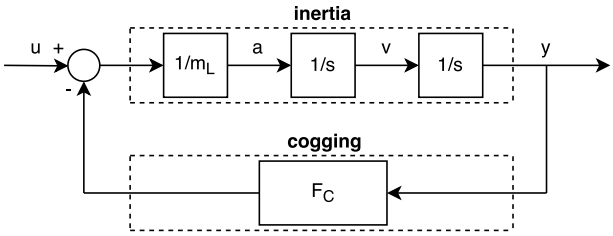


Figure 5.2: Non-ideal plant model including the inertia of the mass and cogging

5.1.1 Design

The 2-dimensional state-indexed LFFC uses two single dimensional B-spline networks. Both networks use their own set of parameters, depending on the influence of the plant to learn. By following the five stage step-by-step plan (used in Chapter 4 as well) the upper and lower input values, the learning rate, the number/distribution of B-splines and the degree of the splines are determined for both networks.

Step 1: Input selection of the BSN
Acceleration BSN: In order to learn the inertia of the mass the learning controller is supplied with the reference acceleration (see Section 4.2.1).

Position BSN: The phenomena cogging is position dependent and therefor it is appropriate to select the reference position as the input for this BSN.

Step 2: Selection of the B-spline order
As was used before, first degree (second order) B-splines will be used in both BSNs. This type of B-splines enable smooth enough function approximation while having the computational time minimized.

Step 3: Selection of the BSN trainings order and trainings motion
The largest influence is caused by the inertia of the mass and is therefor chosen to be learned first. The trainings motions must be selected in such a way that one BSN only observes the dominant behaviour that has to be learned by that BSN. The influence of the other BSN is minimized.

After the acceleration BSN has converged, this knowledge is used to let the position BSN converge. The partial cubic motion is used as a trainings motion for both BSNs (using different parameters).

Acceleration BSN: Dominant behaviour caused by the inertia can be achieved by supplying a reference motion with large accelerations. To minimize the effect of the cogging the motion is repeated five times, by this the effect of cogging is more likely to average out. The motion used is shown in Figure 5.3 in which the maximum acceleration is set to 4.5 [m/s^2] and the range of the forward (and backward) motions is 0.5 [m].

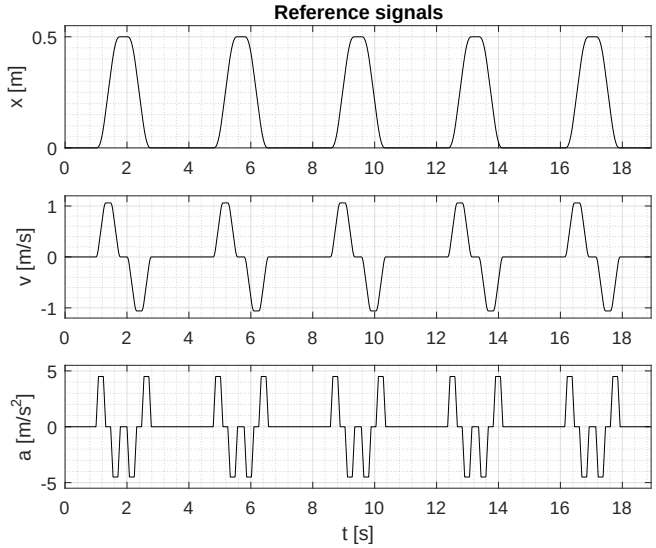


Figure 5.3: Trainings motion used to learn the mass of the plant (acceleration BSN)

Even better performance can be achieved if the parameters of the reference motion supplied will vary randomly. Such as the jerk time, start- and end-time of the motion and the maximum accelerations. Though, this option is not applied to the simulations that follow.

Position BSN: To minimize the influence of the acceleration and make the influence of the cogging dominant a motion must be selected having a low acceleration and a low velocity. The reference motion used is shown in Figure 5.4 in which the maximum accelerations is set to 0.03 0[m/s^2] and the velocity to 0.029 [m/s]. The motion is performed along the whole range onto which the cogging is defined, i.e. from 0 to 0.5 [m].

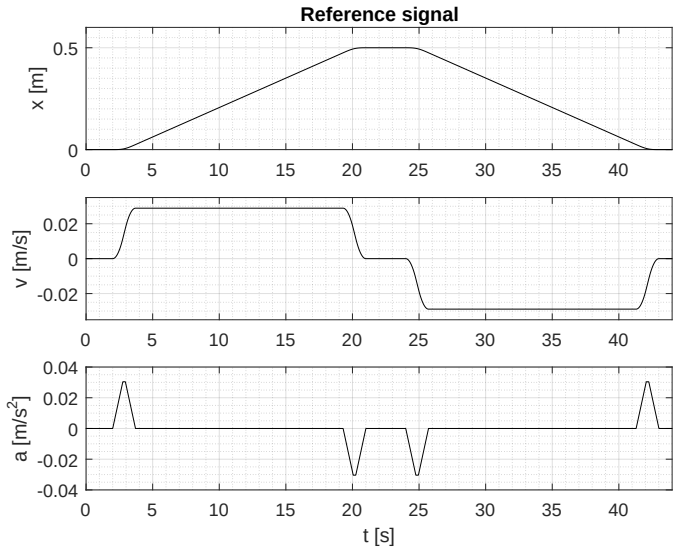


Figure 5.4: Trainings motion used to learn the influence of cogging (position BSN)

To diminish the influence of the acceleration, it is recommended to let some of the parameters vary randomly (for instance the maximum acceleration and velocity). This suggestion is not implemented in the simulations that follow.

Table 5.1: Trainings signal parameters 2-dimensional state-indexed LFFC for the acceleration (learn A) and position (learn X) BSNs

Parameter	Value learn A	Value learn X	Unit	Description
T_{rise}	0.786	19.008	s	Rise time
T_{start}	1.000	2.000	s	Start time
T_{stop}	1.786	21.008	s	Stop time
T_{return}	2.000	24.000	s	Return time
T_{end}	2.786	43.008	s	End time
T_p	3.786	44.008	s	Period of signal
T_{total}	18.93	44.008	s	Total motion time
j_{max}	57.276	0.040	m/s^3	Maximum jerk
a_{max}	4.500	0.0304	m/s^2	Maximum acceleration
v_{max}	1.061	0.0289	m/s	Maximum velocity
x_{max}	0.500	0.500	m	Maximum displacement (= stroke)
CV	20	82	%	Percentage Constant Velocity
CA	20	1	%	Percentage Constant Acceleration

In Table 5.1 an overview is given of the parameter settings for the reference motions used for both learning controllers.

Step 4: Selection of the B-spline distribution and the number of splines
Acceleration BSN: The learning controller is able to approximate the mass by using three uniformly distributed B-splines defined over the input domain of the acceleration. The splines are located at -5 (lowerInput), 0 and 5 (upperInput) [m/s^2].

Position BSN: The number of B-splines required to learn the cogging is determined from the input-output mapping of the position dependent cogging, see Figure 5.5.

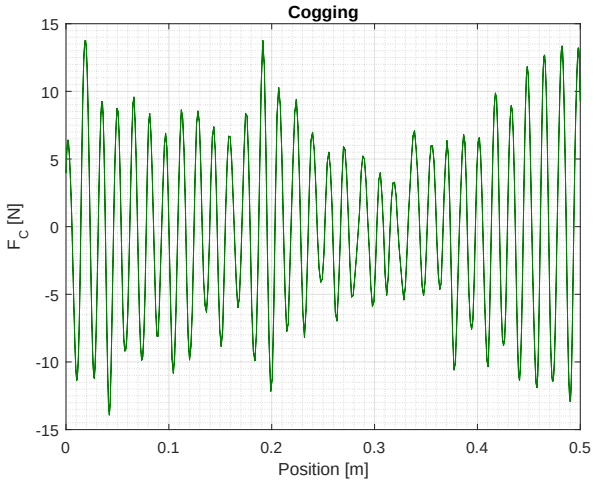


Figure 5.5: Input-output mapping of the position dependent cogging

The mapping of de cogging shows 32 period and is defined along the positions input domain from 0 to 0.5 [m]. To be able to represent the cogging accurately, it is chosen to have about 15 B-splines per period. This can be achieved by selecting 500 B-splines.

Step 5: Selection of learning rate
Mediate learning rates $\gamma = 0.5$ are used for the acceleration and the position learning controllers.

5.1.2 Implementation

The implementation of the 2-dimensional state-indexed LFFC still makes use of the Python Scikit-learn library and the model used in the simulations is shown in Figure 5.6. The model now includes the position dependent cogging as well.

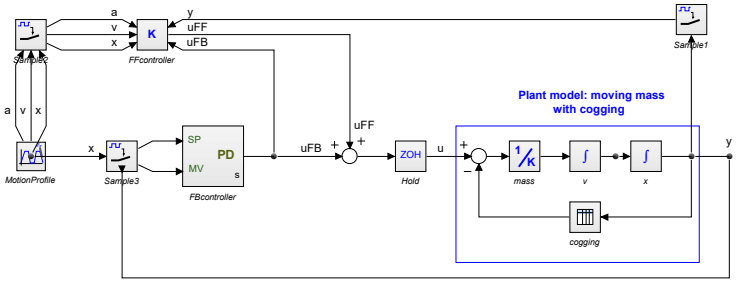


Figure 5.6: 20-sim model used for the 2-dimensional state-indexed LFFC

Within the model the block "FFcontroller" is included that performs the function approximation of the learning controllers and captures the simulation data. Inputs to the block are the signals x, v, a, u_{FB} and y and the output is the signal returned by the function approximator, u_{FF}. The system is discrete and each signal is therefor sampled each 0.001 [s]. A network connection is set-up between 20-sim and Python which enables both programs to communicate (send data back and forth), see Section 3.3.

As was the case for the 1-dimensional LFFC's the function approximation is performed by a B-spline network. The Python functions `interpolate.splrep` and `interpolate.splev` are used to fulfill the function approximation (generate the 3-tuple `(t,c,k)`) and to evaluate the output of the network (generate the output u_{FF}).

The same data processing strategies are used for the acceleration learning controller as was used for the 1D state-indexed LFFC. The input-target mapping is made unique by using the method "average". This method finds the averaged value over the set of duplicated input-target values. The data is sorted and the method "manipulate input-target mapping" is used to match the input-target mapping to the user defined knotvector t. The latter makes sure that in between two consecutive knots an input data point is present.

Furthermore, the input data is checked on values that fall outside the user defined lower and upper input values. The input-target data points that are outside the input domain are removed.

The same data processing procedure is applied to the position learning controller, plus an additional one. As it is known that the reference position should always be positive, the input data is checked on negative values. The input-target point that has a negative input value is removed out of the data set.

In order to let both learning controllers learn independently, the tasks are first set to 1 and 0. The acceleration controller will only be used and start its learning process from zero initial knowledge. After that, the tasks are set to 2 and 1 in which the acceleration controller applies its converged knowledge only and the position controller starts learning from zero initial knowledge.

The 20-sim simulation process can be described according to the five stage communication structure from Section 3.3.3. See Appendix D for more details about the implementation plus an overview of the tasks executed per learning controller.

5.1.3 Simulations

The abilities of the Python implementation of the B-spline network for a 2-dimensional state-indexed LFFC are observed in this section. The feed-forward part consists of two 1-dimensional BSNs. Both BSNs are brought to convergence by performing a multiple run simulation experiment (25 runs) twice, using the 20-sim model shown in Figure 5.6. The performance is checked by observing the tracking error, the learned signals and the convergences of the learning controllers.

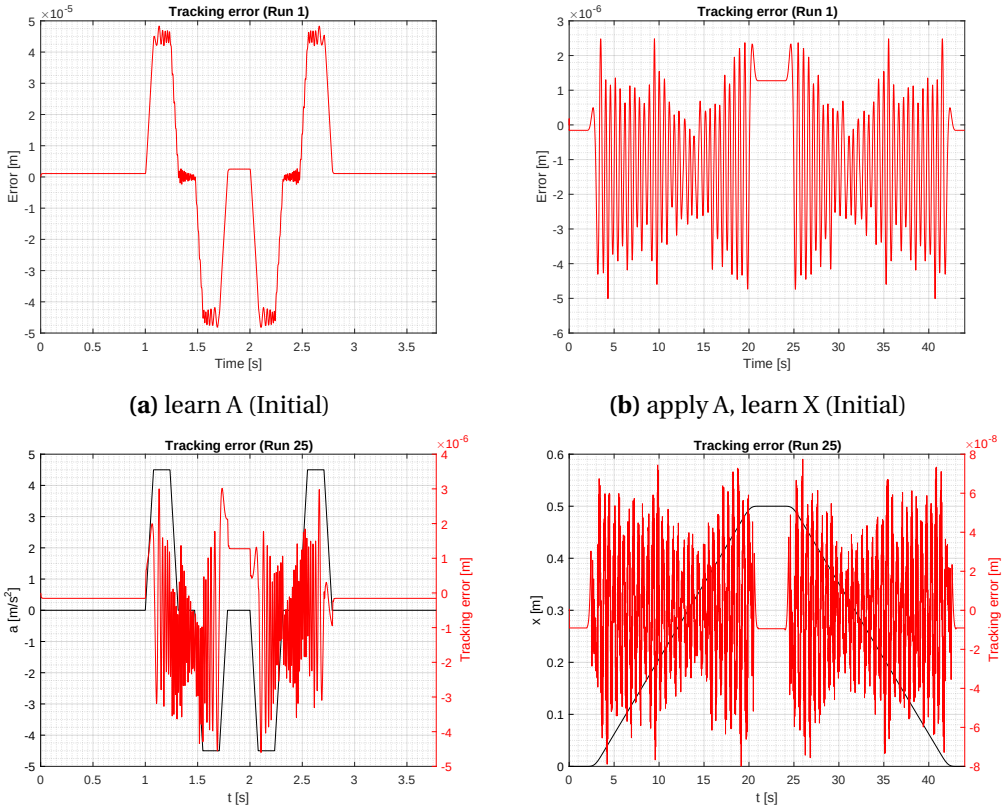
First the acceleration learning controller (LC) is brought to convergence, by setting its task to 1. During this simulation the position learning controller is disabled (task = 0). After that, a new simulation is started in order to let the position learning controller converge. Therefor, the task of the acceleration LC is set to 2 and the position LC to 1. The position learning controller will start learning from zero initial knowledge, while at the same time the acceleration learning controllers applies its converged knowledge about mass m_L.

Different motion profile is supplied to each BSN (see Figure 5.3 and 5.4) and the parameters of the network differ as well. For a summary of the parameters see Table 5.2.

Table 5.2: B-spline parameter settings of Python implementation for a 2-dimensional state-indexed LFFC

Option	**setting learn A**	**setting learn X**
Input (lower)	-5 [m/s^2]	0 [m]
Input (upper)	5 [m/s^2]	0.5 [m]
Learning rate	0.5 [-]	0.5 [-]
Spline order	2 (= degree 1) [-]	2 (= degree 1) [-]
Evaluation order	0 [-]	0 [-]
Number of splines	3 [-]	500 [-]

In Figure 5.7 the tracking errors are shown (together with the reference signals applied to the BSN). The column to the left shows the errors that correspond to the acceleration learning controller. The error that corresponds to Run 1 shows the error the moment no feed-forward is supplied yet. The influence of the inertia of the mass and the cogging can both be distinguish. The influence of the cogging is mainly observed at the segments that represent the constant velocity, but the influence also seems to present at the constant acceleration part.



(a) learn A (Initial)

(b) apply A, learn X (Initial)

(c) learn A (Converged)

(d) apply A, learn X (Converged)

Figure 5.7: Tracking error of Run 1 (initial) and Run 25 (converged). Column left: converge acceleration BSN, don't use position BSN. Column Right: converge position BSN, apply converged acceleration BSN

After the acceleration learning controller has converged the absolute maximum error is reduced from $4.833 \cdot 10^{-05}$ towards $4.600 \cdot 10^{-06}$. The effect of the inertia can no longer be observed for Run 25. The approximation towards this controller converges is 36.3 [kg] (see Table 5.3). As a result of the cogging, this value is 0.7 [kg] lower than in the case without cogging. From run 10 on it can be said that the acceleration controller has converged, at that point the mass approximation is 36.2 [kg]. Figure 5.8 shows the mass approximation per run.

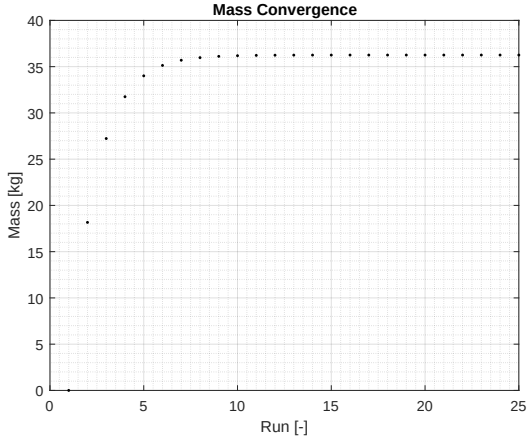


Figure 5.8: Mass approximation per run of the 2-dimensional learning controller

Table 5.3: Mass approximation after 25 runs

Mass approximation (Run 25)	
	learn A
Increasing Decreasing Average	36.254 [kg]

The position learning controller is now brought to convergence, by using the knowledge from the converged acceleration controller. The tracking error that corresponds to the learning process of this controller is shown in the left column of Figure 5.7. In Run 1 the error is shown that is observed after applying the knowledge of the converged acceleration LC and the not yet used position learning controller. The influence of cogging is clearly visible, while the influence of acceleration not.

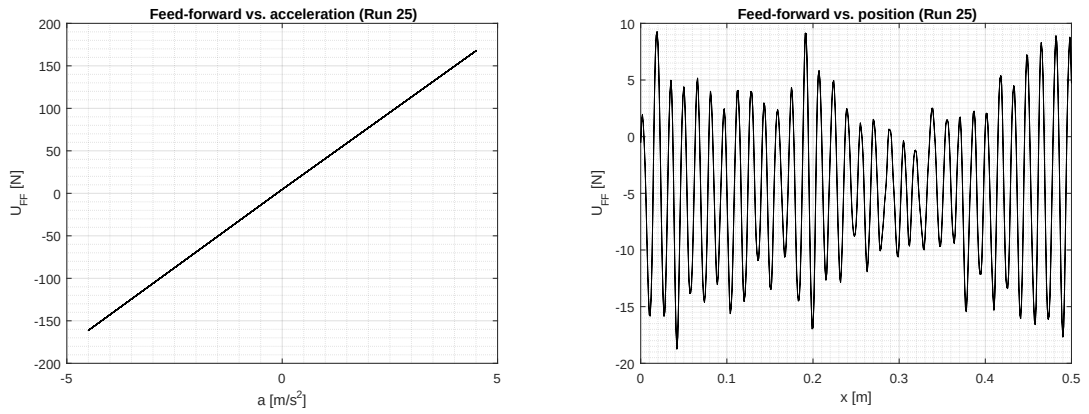
Run 25 shows the converged position learning controller. The tracking error is reduced from $5.009 \cdot 10^{-06}$ towards $7.990 \cdot 10^{-08}$, which is dropped by a factor 60. The influence of cogging can still be observed as the shape is similar to the one observed for Run 1. Increasing the number of B-splines, might result in a lower tracking error but may also introduce some instable behavior.

An overview of the tracking error, after converging each BSN is listed in Table 5.4.

Table 5.4: Absolute mean and maximum error observed for the 2-dimensional LFFC, after convergence of both 1D-BSN

	learn A		**apply A learn X**	
Error	**Run 1 [m]**	**Run 25 [m]**	**Run 1 [m]**	**Run 25 [m]**
Abs. maximum	$4.833 \cdot 10^{-05}$	$4.600 \cdot 10^{-06}$	$5.009 \cdot 10^{-06}$	$7.990 \cdot 10^{-08}$
Abs. mean	$1.191 \cdot 10^{-05}$	$7.814 \cdot 10^{-07}$	$1.551 \cdot 10^{-06}$	$2.277 \cdot 10^{-08}$

To see if both controllers converge to what we expected the learned signal of the corresponding BSN is plotted against its input signal. The acceleration BSN has to learn the mass of the plant and therefor should show a straight line in which the slope represents the approximation of the mass. The position BSN should have learned the cogging and therefor should show a signal that is similar to the input-output mapping of the cogging. The signals learned by both controllers are shown in Figure 5.9.



(a) converged A **(b)** converged X

Figure 5.9: Learned signal by the acceleration BSN (after convergence of A) and position BSN (after convergence of X, using converged data of A)

Both learned signals are according to the approximations. The slope of the figure to the left shows the approximation for mass m_L and the figure to the right shows a signal similar to the cogging (32 periods are visible). That the converged acceleration learning controller has been influenced by the cogging can be observed by taking a close look at the learned signal zoomed-in around zero, see Figure 5.10.

In a completely ideal situation the line would passes exactly through the origin (0,0), such that for $a = 0$ [m/s^2] a zero value for u_{FF} [N] is obtained. Actually, this is not the case. For $a = 0$ [m/s^2] a feed-forward signals if observed of $u_{FF} = 4.580$ [N]. In Chapter 4 similar result was shown and it was obtained that the feed-forward signal for zero acceleration was 0.014 [N].

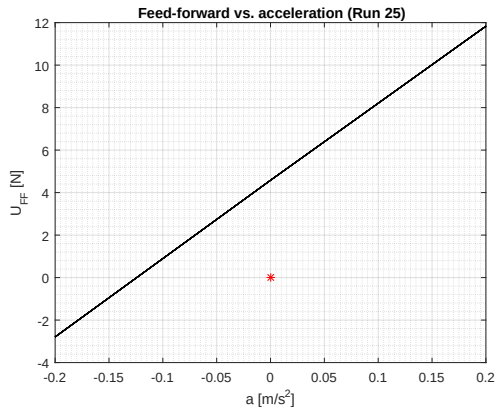


Figure 5.10: Feed-forward signal plotted against acceleration zoomed-in around zero, "*" marks origin (0,0), after converged A

The position learning controller was able to reduce the tracking error and therewith to increase the performance of the system. But, it was not able to perfectly learn the influence. When comparing the learned version of the cogging with the original cogging (Figure 5.5) it can be seen that the maximum amplitude is about 6 [N] too low.

5.1.4 Conclusion

The simulations showed that the 2-dimensional state-indexed LFFC, using two 1-dimensional B-spline Networks showed good performance. By training both BSNs individually the selection for a proper trainings signal was realized. First the acceleration learning controller was brought to convergence by a trainings signal with large acceleration in order to make the influence of the plant by its inertia dominant. After that, the position controller was converged by supplying a trainings signal with low acceleration and low velocity such that the influence of the inertia became low.

The acceleration learning controller was able to approximate mass m_L by deviating 0.5 [kg]. The positing learning controller converged and improved the performance of the control system, even though the obtained maximum amplitude of the input-output mapping of the cogging deviates 6 [N] with the cogging signal defined in the plant model. The controlled system that uses both converged learning controllers showed a drop in absolute maximum tracking error from initially $4.833 \cdot 10^{-05}$ [m] towards $7.990 \cdot 10^{-08}$ [m].

The performance of the system can be improved (lowering the tracking error and converged learning signals closer to the defined signals) by applying random inputs to both learning controllers (explained in Section 5.1.1).

5.2 Multidimensional BSN

In the previous section a 2-dimensional LFFC was shown using two 1-dimensional B-spline networks. In this section a preview is given of a 2-dimensional LFFC using one 2-dimensional BSN. This section is short and treats only some aspects of the LFFC (due to a lack in time for the thesis).

The BSN has two inputs and one output. The inputs supplied are the position x and the acceleration a. The structure of such an LFFC is shown in Figure 5.11.

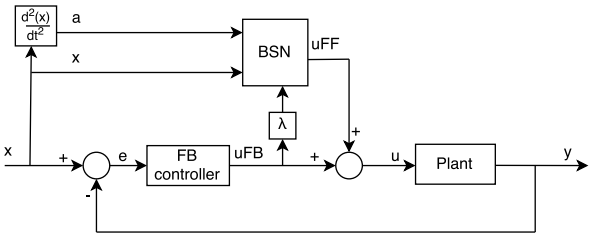


Figure 5.11: Structure of a 2-dimensional state-indexed LFFC (1x 2D-BSN)

The influences of the plant onto the control system has to be learned by the BSN. The influences are the inertia of mass m_L and the position dependent cogging. The plant model is equal to the one used for the 2x 1-dimensional BSN structure and is shown once again in Figure 5.12 .

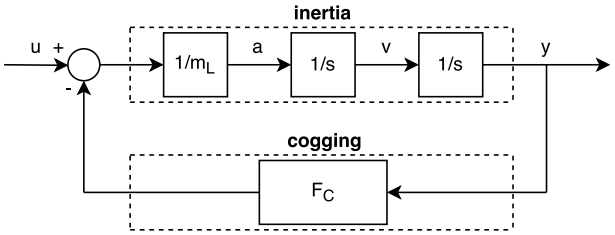


Figure 5.12: Non-ideal plant model including the inertia of the mass and cogging

5.2.1 Design

The design of a 2-dimensional BSN can be done according the same step-by-step plan as was used before. Though, some special attention needs to be paid to the curse of dimensionality.

Step 1: Input selection of the BSN
The inputs chosen are equal to the 2x 1-dimensional BSN:
Acceleration BSN: reference acceleration.

Position BSN: reference position.

Step 2: Selection of the B-spline order
First degree B-splines (second order) are used, as was used for the 2x 1-dimensional BSN.

Step 3: Selection of the BSN trainings order and trainings motion
Actually, for the 1x 2-dimensional BSN there is no specific training order as only one B-spline network has to learn both influences of the plant. In this part of the design stage the curse of dimensionality already starts to get involved. In order to properly train the BSN multiple trainings signals must be presented. The knots of inputs x and a, within the knotvectors tx and ta form together with the network coefficients c a surface: $f(x,a) = c$. A properly trained BSN has observed at each possible point of the surface $x[i], a[j]$, for which $i \in [tx[0], t[i]$ and $j \in [ta[0], ta[j]]$ at least one (preferably more) time a trainings signal.

It can thus be seen that it can be time consuming and perhaps challenging to come up with trainings signals that present each spot. The total number of spots to be reached is: $i * j$, see Figure 5.13.

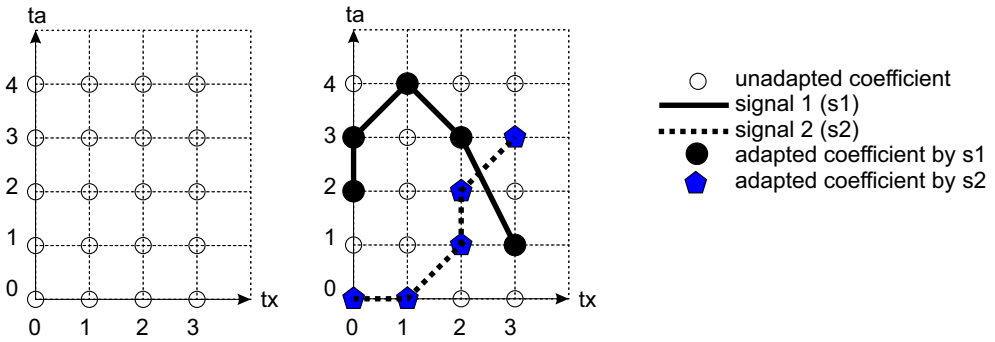


Figure 5.13: 2-dimensional surface representation of $f(x,a) = c$ for which x and a the knots of the position and acceleration learning controller and c the corresponding coefficient (weight). Left figure: untrained BSN and right figure: BSN trained by two signals

Step 4: Selection of the B-spline distribution and the number of splines
The B-spline distribution and the number of B-splines is kept equal to the 2x 1D-BSN:
Acceleration BSN: 3 uniformly distributed B-splines, located at -5, 0 and 5 [m/^{2}].

Position BSN: 500 uniformly distributed B-splines along the range from 0 to 0.5 [m]

Step 5: Selection of learning rate
Mediate learning rate was used for the 2x 1D-BSN and so will be used for the 1x 2D-BSN: $\gamma = 0.5$.

An extra remark is made about the curse of dimensionality. The number of B-splines of the inputs are selected to be $tx = 500$ and $ta = 3$ the total number of basis functions required is:

$$\begin{aligned} N &= N_A \cdot N_X \\ &= 3 \cdot 500 = 1500 \end{aligned} \tag{5.2}$$

Compared to the 2x 1D-BSN, which used only $3 + 500 = 503$ splines this is quite a difference. The larger number of B-splines the more memory is required from an embedded computer. This can be problematic in some cases, especially if the number of B-splines is required to be even more (for instance as more complex influences needs to be represented by the BSN).

The amount of computer memory required is not the only tricky feature of a 2-dimensional (and multi-dimensional) BSN. The generalizing ability of such a BSN is poor. This means that the BSN is not able to produce a feed-forward signal for test signals that were quite similar (but not exactly equal!) to the signals supplied during the training.

Assume a 2-dimensional knot space of the BSN with $tx = 4$ and $ta = 5$. The BSN is partly trained by applying two trainings signals. The total number of coefficients to be trained are $tx * ta = 20$. But, the trainings signals only adapted 10 knots, see Figure 5.14.

Supplied to the BSN is now a test signal. As can be see is that only part of the test signal crosses to the adapted knots during the training. This means that only at those spots ((0,0), (0,2) and (2,3)) the BSN will output a value that has been subjected to the learning of the BSN.

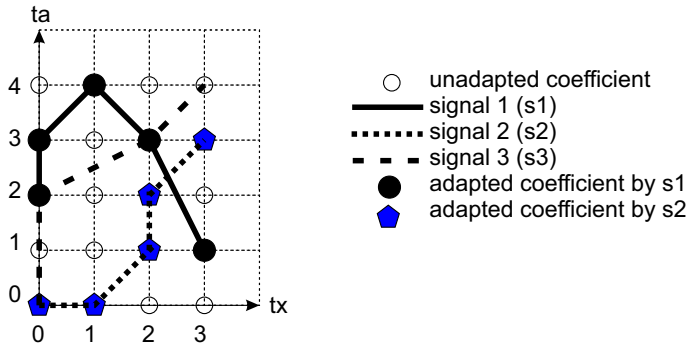


Figure 5.14: Poor generalization of the 1x 2D-BSN

5.2.2 Implementation

This section about the implementation of the 1x 2D-BSN is incomplete and more research has to be done about this topic. The 20-sim model that can be used can be kept equal to the one shown in Figure 5.6 and the same data is send over the network connection between 20-sim and Python (x, v, a, u_{FB} and y). The block "FFController" returns the feed-forward signal u_{FF}.

At the Python side the function `interpolate.bisplrep` and `interpolate.bisplev` can be used. The function approximation is performed by `splrep` which presents a bivariate B-spline of a surface $c = f(x,a)$. The function is supplied with the input arrays x, a and the target signal u_{target}. Knotvectors can be specified for both domains by supplying tx and ta, furthermore the degree of the B-splines needs to be given. Returned is a 5-tuple containing the knotvectors, the coefficients c and the degree of the B-splines for kx and ka.

The output (signal u_{FF}) of the BSN is evaluated by `bisplev`. The evaluation is formed by the cross-product between both rank-1 arrays x and a. Supplied to this function are the arrays x and a and the the 5-tuple `[tx,ta,c,kx,ka]` returned from `bisplrep`.

Restrictions are observed for `interpolate.bisplrep`:

- → inputs must be rank-1 arrays
- → lengths of rank-1 arrays must be equal: $len(x) = len(a) = len(u_{target})$
- → for the degree of the B-splines must hold: $1 \geq kx \leq 5$ and $1 \geq ka \leq 5$
- → a restriction is set for maximum number of data points m
  (the exact number requires more in depth research)

So far, one restriction is found for `interpolate.bisplev`:

- → inputs arrays must be rank-1 arrays

Further research is required in order to set-up a good simulations experiment and to see how well a 2D-BSN is applicable.

6 Conclusion

6.1 Conclusion

In this thesis, a network layer that uses ZeroMQ protocols and Google protocol Buffers for (de-) serialization was designed and tested. Therefor a network layer was added in between the feed-forward part and the general feedback control system. The developed network layer enables the implementation of an LFFC by means of the Python Scikit-learn library in both simulation and experiments.

The comparison between a time-indexed LFFC using 20-sim and Python showed no significant difference in function approximation performances for the function to which the learning controllers converge and the rate at which convergence is observed by using equal learning rates. The absolute maximum tracking error is lower for 20-sim than Python, although the differences are small. A phenomena what appeared in Python only is a time delay, which causes to have different mass approximations for equal accelerations. Further research has to be done about the exact cause, such that this can be resolved.

Based on the control systems performance both systems operate competitively, but the moment user-friendliness and the user's freedom in designing learning controllers are compared Python definitely scores better. Python enables to adjust more (and more easily) the function approximator parameters and a multiple run simulation experiment can be executed automatically.

Simulation results for the 1-dimensional state-indexed LFFC (in Python) showed improved performance the moment the learning controller has converged (learning the mass). Within the first 10 runs of the multiple run simulation experiment convergence appeared and the absolute maximum tracking error dropped by a factor 50.

The 2-dimensional parsimonious LFFC implemented in Python performed well. First the acceleration BSN was trained to compensate for the inertia, the learning controller converged to approximating a mass of 36.5 [kg]. This is only a deviation of 0.5 [kg] from m_L. By using the converged data of the acceleration BSN the position BSN was trained to compensate for the cogging. The learning controller was able to learn the cogging with a deviation in maximum amplitude of 6 [N]. By including this type of LFFC a drop in maximum absolute tracking error was obtained from $4.833 \cdot 10^{-05}$ [m] towards $7.990 \cdot 10^{-08}$ [m].

From the simulations, it can be concluded that the BSN library from Python for 1-dimensional BSNs is promising in its use. A conclusion cannot be drawn about the 2-dimensional BSN library, as more research is required about the implementation of this. What can be said is that using a multidimensional BSN is not preferred as the curse of dimensionality will play a role.

The Scikit-learn library is useful in performing the function approximator (B-spline Network) within the learning controllers of the control systems. This type of approximation combines fast and accurate learning with small computational cost, Brown and Harris (1994). Besides the BSN function approximator, other function approximators present in the Scikit-learn toolbox can be readily applied as well. Those should be considered especially for situations in which multidimensional approximation is needed.

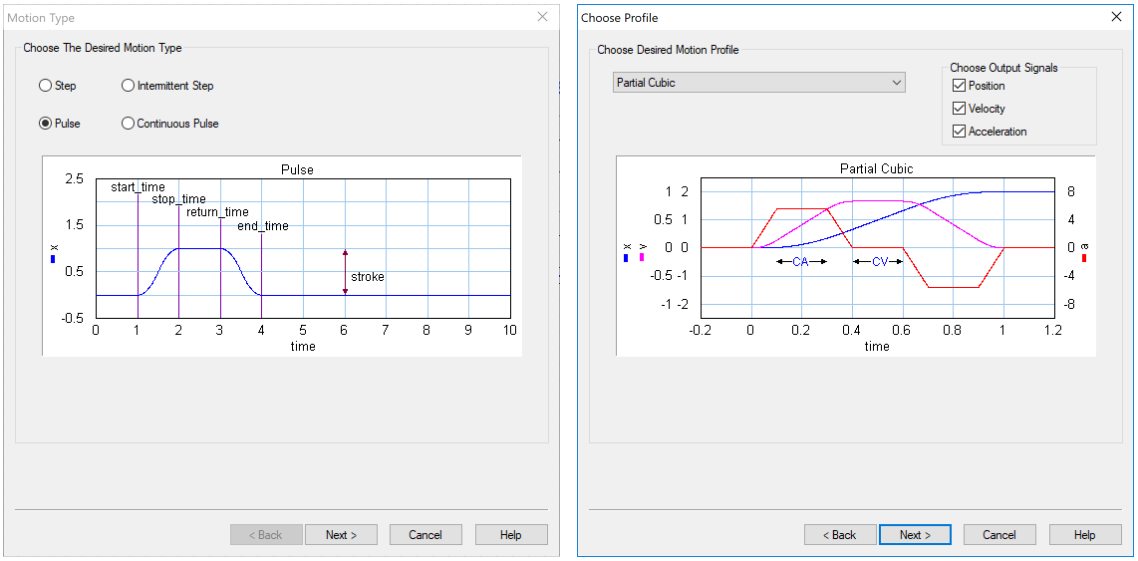
6.2 Future Work

Based on the conclusion and the ambiguities observed during the thesis, the following actions can be performed in future work:

- Fix the bug that causes the 20-sim to crash as soon as the simulation is completed (occurs in simulations in which Python performs the function approximation).
- Perform research about the 2-dimensional state-indexed LFFC that uses one 2D B-spline network and verify its applicability by simulations.
- Re-develop the function approximation part of Python (the functions `splrep` and `bisplrep`) such that it is suitable for use in real-time applications.
- Evaluate the applicability of the 1- and 2- dimensional state-indexed LFFCs when used in a real-world set-up.
- Extend the Python implementation such that function approximators other than Python can also be used.

A Partial Cubic Motion Profile (20-sim)

In Figure A.1 the parameters involved in the definition of a partial cubic reference signal (for position x, velocity v and acceleration a) are shown.



(a) **(b)**

Figure A.1: Partial cubic motion, user window 20-sim

B More about B-splines

B.1 Properties of B-spline Basis Functions

The set of basis functions, $N_{i,p}(u)$ that is described by Equations 2.5 and 2.6 have the following properties:

- polynomial in u having degree p
- non-negative for all i, p, and u (**non-negativity**)
- non-zero on the half open interval $[u_i, u_{i+p+1})$ (**local support**)
- at most $p+1$ degree p basis functions are non-zero in the knot span defined on the half open interval $[u_i, u_{i+1})$: $N_{i-p,p}(u), N_{i-p+1,p}(u), N_{i-p+2,p}(u), ..., N_{i,p}(u)$
- for $m+1$ knots, basis functions of degree p and $n+1$ basis functions, the following equation is satisfied: $m = n+p+1$
- constructed from parabolas of degree p, and each parabola is connected at the knots $[u_i, u_{i+1})$ (**composite curve**)
- is C^{p-k} continuous for a knot having multiplicity k

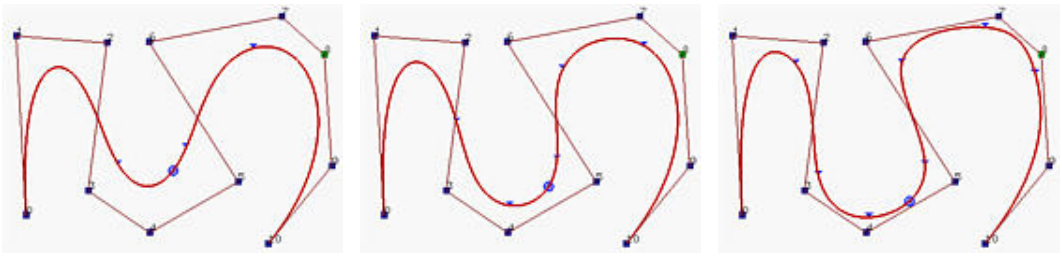
Multiple knots will have the following impact on the computation of the basis functions:

1. Knots with multiplicity k will affect $k-1$ basis functions. For each increase in multiplicity a knot span will disappear, so for $k = 1 \rightarrow$ one span, $k = 2 \rightarrow$ two spans disappear, and so on.
2. Knots with multiplicity k have at most $p-k-1$ non-zero basis functions. For each increase in multiplicity of a knot will reduce the number of non-zero basis functions of that knot

B.2 Properties of B-spline Curves

In defining the properties of B-spline curves it is assumed that B-spline curve $\mathbf{C}(u)$ is of degree p, is defined by $n+1$ control points and the curve domain is defined by knotvector $U = [u_0, u_1, ..., u_m]$. The curve is of the clamped type which means that $u_0 = u_1 = ... = u_p$ and $u_{m-p}, u_{m-p+1} = ... = u_m$. The most important properties are described below:

- clamped $\mathbf{C}(u)$ is a piecewise curve and on each segment a curve of degree p exists. As a result, it is possible to design more complex curves while using lower polynomial degrees (when compared to Bézier curves). The lower the degree the closer a curve can follow its control line, in general. In Figure B.1 the curve fitting is shown for B-splines having degree 7, 5 and 3, while using the same control polyline.



(a) **(b)** **(c)**

Figure B.1: B-spline curve and its control polyline for: a) degree 7, b) degree 5 and c) degree 3

- $m = n+p+1$ must be satisfied, it implies that each control point requires a basis function
- clamped $\mathbf{C}(u)$ passes through the control endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$. The coefficient of control point $\mathbf{P}_0$ is the basis function $N_{0,p}(u)$ and is non-zero on the half open interval $[u_0, u_{p+1})$. Since $u_0 = u_1 = ...u_p = 0$ for the clamped B-spline curve the coefficients $N_{0,0}(u), N_{1,0}(u), ..., N_{p-1,0}(u) = 0$ and only $N_{p,0}$ is non-zero such that for $u = 0 \rightarrow N_{0,p}(0) = 1$ and $\mathbf{C}(0) = \mathbf{P}(0)$. Similar description can be given for $\mathbf{C}(1) = \mathbf{P}(n)$.
- The B-spline curve is contained in the convex hull of its control polyline (**convex Hull property**). For u in knot span $[u_i, u_{i+1})$ only $p+1$ basis functions are non-zero, i.e. $N_{i,p}(u), ..., N_{i-p+1,p}(u), N_{i-p,p}(u)$. $\mathbf{C}(u)$ is the convex hull of control points $\mathbf{P}_{i-p}, \mathbf{P}_{i-p+1}, ..., \mathbf{P}_i$.

  For example, assume u is in knot span $[u_9, u_{10})$ and the non-zero basis functions are $N_{9,3}(u), N_{8,3}(u), N_{7,3}(u)$ and $N_{6,3}(u)$ having corresponding control points $\mathbf{P}_9, \mathbf{P}_8, \mathbf{P}_7$ and $\mathbf{P}_6$. In Figure B.2 it is shown that $\mathbf{C}(u)$ lies in the convex hull of its control points (shaded area).

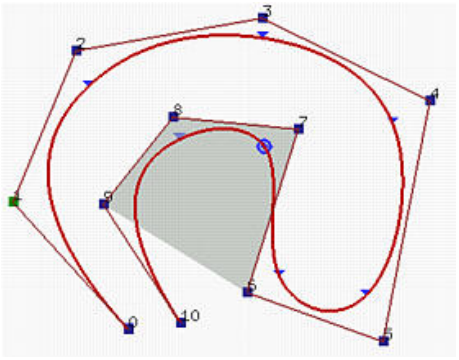


Figure B.2: Convex Hull property of B-spline curve

- A curve can be locally modified without changing its global shape, see Figure B.3 in which control point $\mathbf{P}_2$ is moved (**local modification scheme**).

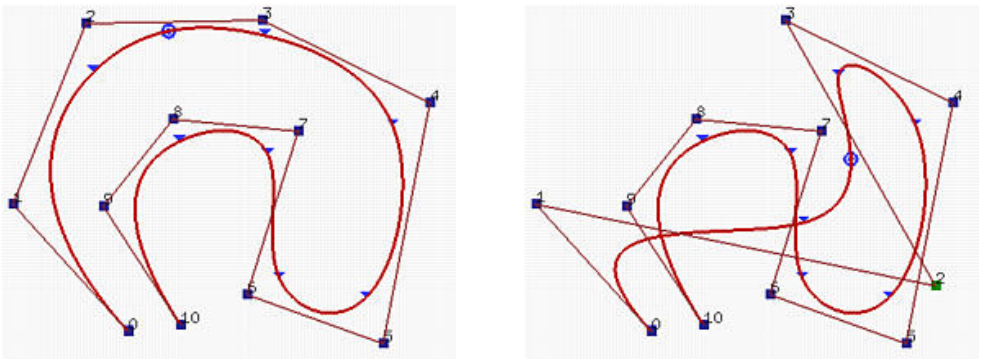


Figure B.3: Local modification property of B-spline curves, right figures shows the effects of moving control point $\mathbf{P}_2$

- Clamped $\mathbf{C}(u)$ is C^{p-k} continuous at a knot with multiplicity k. In the case C^0 the curve passes through a control point, for C^1 the corresponding point lies on a leg. Continuities larger than 1 are more difficult to express (or visualize) the differences.
- For a curve being in a plane, it is not possible to draw a straight line that intersects a B-spline curve more times than that it intersects the control polyline of the curve (**variation diminishing property**).
- If the degree of a B-spline is equal to n, so $p = n$ and one less than the number of control points, the curve is clamped $p+1$ at each end, the B-spline curve reduces to a Bézier curve: $2(p+1) = 2(n+1)$
- The result of applying an affine transformation to a B-spline curve can be constructed from the affine images of its control points (**affine invariance**). This way an affine transformation can be applied to the B-splines control points in stead of to the curve itself.

B.2.1 Moving Control Points

The shape of a B-spline curve can be changed by moving its control points, Mtu (2017). The change of control point $\mathbf{P}_i$ will only affect curve $\mathbf{C}(u)$ on the interval $[u_i, u_{i+p+1})$. So, in moving control point $\mathbf{P}_i$ to certain position $\mathbf{Q}_i$ the curve $\mathbf{C}(u)$ will move in the same direction. But its position might differ from point to point. A translation of control point $\mathbf{P}_i$ by v to $\mathbf{Q}_i$, can be described by the sum of the original curve $\mathbf{C}(u)$ and a translational vector $N_{i,p}(u)\mathbf{v}$. This results in the new curve $D(u)$:

$$\mathbf{D}(u) = \mathbf{C}(u) + N_{i,p}(u)\mathbf{v} \tag{B.1}$$

To illustrate the movement of a control point and its effect an example is given. In Figure B.4 (left) a B-spline curve of degree $p = 4$ is shown. This curve is defined by 13 control points ($n = 12$) and 18 knots $m = 17$. The knots are simple and the curve is clamped, i.e. $u_0 = u_1 = u_2 = u_3 = u_4 = 0$ and $u_{13} = u_{14} = u_{15} = u_{16} = u_{17} = 1$. The remainder are 9 knots spans defining 9 segments as shown in Table B.1.

Table B.1: Knot span interval for segments 1 till 9

span	segment
$[u_4, u_5)$	1
$[u_5, u_6)$	2
$[u_6, u_7)$	3
$[u_7, u_8)$	4
$[u_8, u_9)$	5
$[u_9, u_{10})$	6
$[u_{10}, u_{11})$	7
$[u_{11}, u_{12})$	8
$[u_{12}, u_{13})$	9

Moving control point $\mathbf{P}_6$ downwards (see Figure B.4 right) results in a movement of the curve in the same direction. The coefficient of $\mathbf{P}_6$ is $N_{6,4}(u)$ and only the non-zero interval $[u_6, u_{11})$ of the curve will be affected, i.e. only segments 3, 4, 5, 6 and 7.

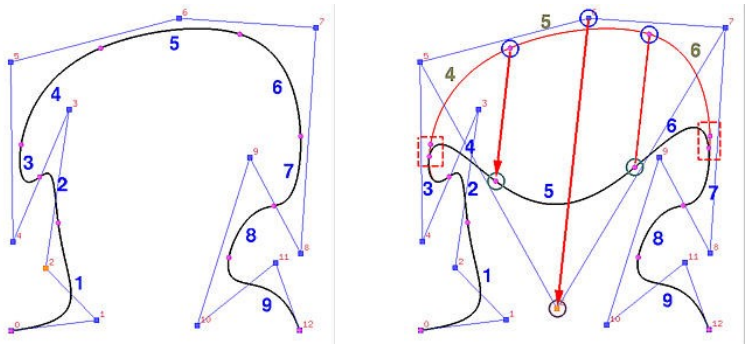


Figure B.4: Moving a control points and its influence on the B-spline curve

B.2.2 Modifying Knots

A B-spline curve consists of several curves segments linked together. Each segment is defined by a knot span. By changing the position of on or more knots, the knot span of the corresponding curve segment changes and therewith the shape of the curve.

In order to modify the shape of a curve by changing the position of knots and to guarantee the curve change the way as expected, it is wise to create internal knots that have a multiplicity of k. The moment the multiplicity of a knot increases the number of non-zero basis functions decreases. For a knot having multiplicity k there are at most $p-k+1$ non-zero basis functions. Keeping this property in mind, three situations can be drawn by changing the multiplicity of a knot:

- **Multiplicity:** $p-k$
  There are $k+1$ non-zero basis functions at the knot and the point lies within the convex hull defined by the control points that correspond to those non-zero basis functions
- **Multiplicity:** $k = p-1$
  There are two non-zero basis functions at the knot and the convex hull is a line segment
- **Multiplicity:** $k = p$
  There is only one non-zero basis function at the knot and only one control point having a non-zero coefficient. The curve passes through this point

C Implementation details (1D state-indexed LFFC)

The tasks make a distinction in the learning controllers feature to learn and/or to apply previously obtained knowledge.

Table C.1: Tasks the learning controller (LC) can perform

Option	Learn	Apply	Description
`Task = 0`	×	×	LC will not be used
`Task = 1`	✓	×	Learning starts from zero previous knowledge
`Task = 2`	×	✓	LC only applies previous knowledge specified by user
`Task = 3`	✓	✓	Start learning from previous knowledge specified by user

The implementation of the 1-dimensional learning controller is explained using the five stage communication structure between 20-sim and Pyhon (see Section 3.3.3). Compared to the time-indexed LFFC at the start of the simulation the user input is requested to specify the files containing the learning controller task and the B-spline parameter settings (both in `.txt` format).

Furthermore, the user is able to select a folder in which the simulation data will be stored. A multiple run simulation experiment is assumed in order to explain the five stage communication structure, see Table C.2. Depending on the task some functions will or will not be executed, in Table C.3 this is shown.

Table C.2: Simulation stages and its actions, for a multiple run simulation experiment using a time-indexed LFFC

Stage 1:	`Initialize()`
→	Set-up a network connection between 20-sim and Python
→	Set `runNumber` = 0 (run number in simulation)
→	User input: select `.txt` file containing the learning controllers task
→	User input: select folder in which simulation data will be stored
→	Create feedback folders for data storage
→	Create learning folder "acceleration" for data storage
→	User input: select `.txt` file containing the B-spline parameter settings
→	Read parameters selected parameter file:
	`lowerInput`: lower value of BSN input domain
	`upperInput`: upper value of BSN input domain
	`numberOfKnots`: number of B-splines distributed over BSN input domain
	`splineDegree`: degree of the B-splines
	`evalDegree`: degree of the output evaluation
	`learningRate`: learning rate of the BSN
→	Determine knotvector and internal knots
→	Select initial `(t,c,k)` file as starting knowledge
→	Create initial 3-tuple `(t,c,k)`, with c all zeros
→	Save initial 3-tuple `(t,c,k)`
Stage 2:	`InitializeRun()`
→	Increase value of `runNumber` by 1
→	Clear feedback arrays
→	Clear learning arrays
→	Load and store tuple `(t,c,k)` from the previous run
	(or the initialized one, if `runNumber=1`)
→	Create initial data set, with data point in between two consecutive knots
Stage 3:	`ProcessStep()`
→	Append feedback arrays: u_{FB}, x, v, a and y
→	Append input array (a)
→	Evaluate the output value $u_{FF}[i]$ using `interpolate.splev`
→	Determine target value: $u_{target}[i] = u_{FF}[i](previousrun) + u_{FB}[i] \cdot \gamma$
→	Append arrays: $u_{target}[i]$ and $u_{FF}[i]$ returned from `interpolate.splev`
Stage 4:	`TerminateRun()`
→	Save to file feedback arrays: u_{FB}, x, v, a and y
→	Match input to B-spline parameters
→	Compare current input-target data with data set of previous run
→	Filter duplicated data
→	Sort input-target mapping in ascending order
→	Save to file: input array (a) and feed-forward signal array u_{FF}
→	Determine `(t,c,k)` for next run using `interpolate.splrep`
→	Store combined input data to be used as old data in next run
→	Save to file: 3-tuple `(t,c,k)` to be used in the next run
Stage 5:	`Terminate()`
→	Close network connection between 20-sim and Python

Table C.3: Functions executed per task of the learning controller (* Only for `runNumber>1` else selected tck file from user input is used)

`initializeSim()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Set-up Python and 20-sim network connection	✓	✓	✓	✓
Set `runNumber=0`	✓	✓	✓	✓
User input: select `.txt` file containing task	✓	✓	✓	✓
Load tasks from file	✓	✓	✓	✓
User input: select data storage folder	✓	✓	✓	✓
Create feedback folders: x, v, a, u_{FB}, y	✓	✓	✓	✓
Create learning folders: input a, feed-forward u_{FF}	×	✓	✓	✓
Create learning folder: tck	×	✓	×	✓
User input: select `.txt` file B-spline parameters	×	✓	✓	✓
Load parameters from file	×	✓	✓	✓
Determine knotvector from parameters	×	✓	×	✓
User input: select `.dump` file previous tck	×	×	✓	✓
Load tck from file	×	×	✓	✓
Create initial tck	×	✓	×	×
Save to file `.dump` initial tck	×	✓	×	×

`initializeRun()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Set `runNumber+=1`	✓	✓	✓	✓
Clear feedback arrays: x, v, a, u_{FB}, y	✓	✓	✓	✓
Clear learning arrays: $input, u_{FFprev}, u_{FF}$	×	✓	✓	✓
Clear learning array: u_{target}	×	✓	×	×
Load tck for `runNumber-1`	×	✓	×	×*
Create initial dataset matching to knotvector (only Run 1)	×	✓	×	✓
Evaluate target value corresponding to initial input data	×	✓	×	✓

`processStep()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Append feedback arrays: x, v, a, u_{FB}, y	✓	✓	✓	✓
Append learning array: input a	×	✓	✓	✓
Evaluate u_{FF} of previous run, use `splev`	×	✓	✓	✓
Determine target $target = u_{FFprev} + u_{FB} \cdot \gamma$	×	✓	×	✓
Append learning array: u_{target}	×	✓	×	✓
Append learning array: u_{FFprev}	×	✓	×	✓
Send to 20-sim u_{FFprev}	✓	✓	✓	✓

`terminateRun()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Save feedback data to file: x, v, a, u_{FB}, y	✓	✓	✓	✓
Match input to B-spline parameters	×	✓	×	✓
Compare/combine input data present and previous run	×	✓	×	✓
Filter duplicates from input-target mapping	×	✓	×	✓
Sort input-target mapping in ascending order	×	✓	×	✓
Save learning data to file: a and u_{FF}	×	✓	×	✓
Determine tck for next run using `splrep`	×	✓	×	✓
Store combined input-target data as old (use next run)	×	✓	×	✓
Save learning data to file: tck	×	✓	×	✓

`terminateSim()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Terminate Python and 20-sim network connection	✓	✓	✓	✓

D Implementation details (2D state-indexed LFFC)

Table D.1: Simulation stages and its actions, for a multiple run simulation experiment using a time-indexed LFFC

Stage 1:	`Initialize()`
→	Set-up a network connection between 20-sim and Python
→	Set `runNumber` = 0 (run number in simulation)
→	User input: select `.txt` file containing the learning controllers task
→	User input: select folder in which simulation data will be stored
→	Create feedback folders for data storage
→	Create learning folder "acceleration" for data storage
→	User input: select `.txt` file containing the B-spline parameter settings
→	Read parameters selected parameter file:
	`lowerInput`: lower value of BSN input domain
	`upperInput`: upper value of BSN input domain
	`numberOfKnots`: number of B-splines distributed over BSN input domain
	`splineDegree`: degree of the B-splines
	`evalDegree`: degree of the output evaluation
	`learningRate`: learning rate of the BSN
→	Determine knotvector and internal knots
→	Select initial `(t,c,k)` file as starting knowledge
→	Create initial 3-tuple `(t,c,k)`, with c all zeros and save to file
Stage 2:	`InitializeRun()`
→	Increase value of `runNumber` by 1
→	Clear feedback arrays
→	Clear learning arrays
→	Load and store tuple `(t,c,k)` of the previous run (or init, if `runNumber=1`)
→	Create initial data set, with data point in between two consecutive knots
Stage 3:	`ProcessStep()`
→	Append feedback arrays: u_{FB}, x, v, a and y
→	Append input array (a)
→	Evaluate the output value $u_{FF}[i]$ using `interpolate.splev`
→	Determine target value: $u_{target}[i] = u_{FF}[i](previousrun) + u_{FB}[i] \cdot \gamma$
→	Append arrays: $u_{target}[i]$ and $u_{FF}[i]$ returned from `interpolate.splev`
Stage 4:	`TerminateRun()`
→	Save to file feedback arrays: u_{FB}, x, v, a and y
→	Match input to B-spline parameters
→	Remove input-target point with negative input values
→	Compare current input-target data with data set of previous run
→	Filter duplicated data
→	Sort input-target mapping in ascending order
→	Save to file: input array (a) and feed-forward signal array u_{FF}
→	Determine `(t,c,k)` for next run using `interpolate.splrep`
→	Store combined input data to be used as old data in next run
→	Save to file: 3-tuple `(t,c,k)` to be used in the next run
Stage 5:	`Terminate()`
→	Close network connection between 20-sim and Python

Table D.2: Functions executed per task of the learning controller (* Only for `runNumber>1` else selected tck file from user input is used)

`initialize()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Set-up Python and 20-sim network connection	✓	✓	✓	✓
Set `runNumber=0`	✓	✓	✓	✓
User input: select `taskLFFC.txt`	✓	✓	✓	✓
Load tasks from file	✓	✓	✓	✓
User input: select data storage folder	✓	✓	✓	✓
Create folders for x, v, a, u_{FB}, y	✓	✓	✓	✓
Create folders for $input, u_{FF}$	×	✓	✓	✓
Create folders for tck	×	✓	×	✓
User input: select tck file to load	×	×	✓	✓
Load tck from file	×	×	✓	✓
User input: select `parameter.txt`	×	✓	✓	✓
Load parameters from file	×	✓	✓	✓
Determine knotvector from parameters	×	✓	×	✓
Create/store initial tck	×	✓	×	×

`initializeRun()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Set `runNumber+=1`	✓	✓	✓	✓
Clear arrays for x, v, a, u_{FB}, y	✓	✓	✓	✓
Clear arrays for $input, u_{FFprev}, u_{FF}$	×	✓	✓	✓
Clear array $target$	×	✓	×	×
Load tck for `runNumber-1`	×	✓	×	×*
Create initial data set	×	✓	×	✓
Evaluate target (`splev`) corresponding to initial data set	×	✓	×	✓

`processStep()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Append arrays for x, v, a, u_{FB}, y	✓	✓	✓	✓
Append arrays for $input, u_{FFprev}, u_{FF}$	×	✓	✓	✓
Append array $target$	×	✓	×	✓
Evaluate u_{FF} of previous run, use `splev`	×	✓	✓	✓
Determine target $target = u_{FFprev} + u_{FB} \cdot \gamma$	×	✓	×	✓
Sum u_{FFprev} of learning controllers	✓	✓	×	✓
Send to 20-sim u_{FFprev}	✓	✓	✓	✓

`terminateRun()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Save to file x, v, a, u_{FB}, y	✓	✓	✓	✓
Math inputs of reference with BSN inputs	×	✓	×	✓
Remove duplicates from input data	×	✓	×	✓
Order input data ascending	×	✓	×	✓
Determine tck for next run	×	✓	×	✓
Save to file tck	×	✓	×	✓
Save to file $input, u_{FF}$	×	✓	✓	✓

`terminateRun()`	**Task 1**	**Task 2**	**Task 3**	**Task 4**
Terminate Python and 20-sim network connection	✓	✓	✓	✓

Bibliography

(2017a), Python interpolate.bisplev.
https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.bisplev.html

(2017b), Python interpolate.bisplrep.
https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.bisplrep.html

20simBSN (2017), 20sim B-spline Network.
http://www.20sim.com/webhelp/toolboxes_control_toolbox_b-spline_network_editor_introductionbsplinenetworks.php

20simDynamicDLL (2017), 20sim Dynamic DLL.
http://www.20sim.com/webhelp/language_reference_functions_writingdynamicdlls.php

Bishop H. Robert H. Bishop (2007), Mechatronic System Control, Logic, and Data Acquisition, Taylor Amp; Francis Inc, chapter 1, p. 700, 2nd edition, ISBN 978-0-8493-9260-3.

Bossley, K. and C. Harris (1997), *Neurofuzzy Modelling Approaches in System Identification*, Ph.D. thesis, University of Southampton.

Brown, M. and C. Harris (1994), *Neurofuzzy Adaptive Modelling and Control*, Prentice Hall.
http://eprints.soton.ac.uk/id/eprint/250255

Buijssen, S. (2001), Learning feed-forward control applied on the H-drive, Technical report, Technische Universiteit Eindhoven.
https://pure.tue.nl/ws/files/4239441/632766.pdf

Harris, C., C. Hoore and M. Brown (1993), *Intelligent Control: Aspects of Fuzzy Logic and Neural Nets*, world scie edition.

de Kruif, B. J. and T. J. de Vries (2000), IMPROVING PRICE/PERFORMANCE RATIO OF A LINEAR MOTOR BY MEANS OF LEARNING CONTROL, Technical report, University of Twente, Drebbel institute of mechatronics.
http://www.nici.ru.nl/mmm/personal/kruif/publications/IFAC02.pdf

Mtu (2017), B-spline Curves: Moving Control Points.
http://pages.mtu.edu/$\sim$shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-mv-ctlpt.html

O'Flaherty, R. and M. Egerstedt (2015), Low-dimensional learning for complex robots, **vol. 12**, no.1, pp. 19–27, ISSN 15455955, doi:10.1109/TASE.2014.2349915.

Otten, G., T. de Vries, J. van Amerongen, A. Rankers and E. Gaal (1997), Linear motor motion control using a learning feedforward controller, **vol. 2**, no.3, pp. 179–187, ISSN 10834435, doi:10.1109/3516.622970.
http://ieeexplore.ieee.org/document/622970/

Polycarpou, M. M. and P. A. Ioannou (1992), , no.December 1892, pp. 7–12.
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=371802

ProtoBuf (2017), Protocol Buffer from Google (Proto 3).
https://developers.google.com/protocol-buffers/docs/proto3

Scikit-learn (2017), Python Scikit-learn.
http://scikit-learn.org/stable/

Scipy Manual (2017), Scipy interpolation.
https://docs.scipy.org/doc/scipy/reference/interpolate.html

Splev (2017), Python interpolate.splev.
https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.interpolate.splev.html#scipy.interpolate.splev

Splrep (2017), Python interpolate.splrep.
https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.interpolate.splrep.html#scipy.interpolate.splrep

Starrenburg, J., W. van Luenen, W. Oelen and J. van Amerongen (1996), No Title, *Eontrol Eng. Pract.*, **vol. 4**, pp. 1221–1230.
https://ris.utwente.nl/ws/portalfiles/portal/6645346

T.J.A.de Vries, W.J.R.Velthuis and L.J.Idema (2001), Application of parsimonious learning feedforward control to mechatronic systems, *IEE Proc. - Control Theory Appl.*, **vol. 148**.
http://imotec.nl/imotec.nl/wp-content/uploads/2013/09/ApplParsimoniousLFFC2001IEEProcD.pdf

Velthuis, W., T. de Vries, K. Vrielink, G. Wierda and A. Borghuis (1998), Learning Control of a Flight Simulator Stick, pp. 29–34.
https://www.ram.ewi.utwente.nl/aigaion/attachments/single/484

Velthuis, W. J. (2000), *Learning Feed-Forward Control - Theory, Design and Applications*, ISBN 90-36514126.
http://www.ub.utwente.nl/webdocs/el/1/t0000012.pdf

de Vries, T. (2015), Command Response of a Moving Mass.

ZeroMQ (2017), ZeroMQ.
http://zguide.zeromq.org/page:all