Daniel Karner, BSc

Automatic Tuning of Load Unit Observer Parameters for Automotive Testbeds

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Martin Horn

Institute for control and automation technology

Graz, February 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date	Signature

Abstract

This master thesis deals with the automatic parameters identification of a speed observer. The speed observer is part of the control system of an electrical load unit, which is integrated in an engine or powertrain testbed. Previously, this speed observer parameters have been commonly tuned by special trained experts. Parameters that are found by less trained people, are often incorrect, which leads to an unnecessary high commissioning effort and insufficient control behaviour.

The commissioning complexity of an engine or powertrain testbed can be reduced on the basis of the automatic parameter tuning process developed here. Therefore, important time and costs of commissioning can be saved. Furthermore, ideal speed observer parameters ensure good and robust operating and guarantee specified control properties of the automotive testbed. For the developed identification process, corresponding test runs must be performed and recorded on the testbed. Afterwards, the ideal speed observer parameters are identified through software analysing of the recorded files.

Additionally, the identified observer parameters can be adapted manually and they are validated by the identification software via several validation plots. With the help of this standardised tuning mechanism of the speed observer, the parameterisation errors get reduced and thus the quality of the corresponding controlling concept increases.

Zusammenfassung

Diese Masterarbeit befasst sich mit der automatisierten Parameteridentifikation eines Drehzahlbeobachters, der in einem System zur Regelung eines elektrischen Belastungsmotors in einem Motor- oder Antriebstrangprüfstand integriert ist. Bisher wurden die Parameter händisch von speziell ausgebildeten Experten ermittelt. Parameter, die von weniger ausgebildeten Personen gefundenen wurden, waren oft fehlerbehaftet, was zu unnötig hohem Inbetriebnahmeaufwand und unzureichenden Regelungseigenschaften führte.

Durch den im Rahmen dieser Masterarbeit erstellten Identifikationsautomatismus werden die Komplexität der Inbetriebnahme eines Motor- oder Antriebstrangprüfstandes reduziert und somit wertvolle Zeit gespart und Kosten gesenkt. Optimierte Parameter gewährleisten einen idealen und robusten Betrieb des Prüfstandes und stellen die spezifizierten Regelungseigenschaften sicher. Zu diesem Zwecke werden am Prüfstand entsprechende Prüfläufe ausgeführt und Messergebnisse erzeugt, die anschließend von einer Software zur Bestimmung der Beobachterparameter analysiert werden.

Zusätzlich werden die ermittelten Parameter durch Grafiken validiert und können gegebenenfalls manuell angepasst werden. Anhand dieser standardisierten Einstellungsmethode für die Parameter des Drehzahlbeobachters sollen Parametrierfehler vermieden, die Qualität der Parameter erhöht und somit die Regelungseigenschaften verbessert werden.

Contents

1 Introduction

Technologies as well as their requirements are developing very fast. Therefore, it is important to keep up with this technical progress. Whereas previously in control engineering the parameters for a common loop controller would have been set up manually, nowadays, the settings of a whole control system are automatically calculated by an identification algorithm. This automatic tuning does not only save time, it often reaches better control behaviour of the given system than a manual parameterisation. For this reason, the industry is interested in the development of such automatic tuning tools.

This thesis deals with a concept for automatic or rather standardised tuning of a speed observer. The speed observer in question operates in an automotive testbed manufactured by AVL GmbH (*A*nstalt für *V*erbrennungskraftmaschinen *L*ist). The observer provides the revolution speed of electrical machines and is therefore an essential part of the implemented control system.
At the moment, the total parameterisation of the controlling unit is done in an elaborated manual process by a testbed engineer. From that situation, the idea for an automatic tuning concept of the controlling system was born. This automatic tuning mechanism should relieve the testbed engineer and speed up the tuning process as well as improve the controlling performance. Finally, the most important thing for a correctly working closed-loop control, is a well matched configuration of the controlling system unit.

1.1 Automotive Testbeds

As already mentioned above, the speed observer is an important part of the control system of the automotive testbeds. It observes the speed of an electrical machine and provides the calculated signal as the feedback for the testbed controllers. The testbeds, which use this observer, are the engine (see figure 1.1) on the one hand and the powertrain testbeds (see figure 1.2) on the other.
At the engine testbed, an electrical machine simulates the whole environment consisting of car, street, powertrain, etc. by loading or driving the combustion or electrical engine. This makes it possible to test and enhance engines and it is easy to simulate different driving scenarios without any additional resources. Therefore, the automotive testbed enables a simple and efficient way for engineering power units.
At powertrain testbeds, the units under test (abbr. UUT) are individual transmissions with or without a driving engine. Depending on the powertrain type, this testbed exists in different configurations, which will be discussed later in chapter 2.

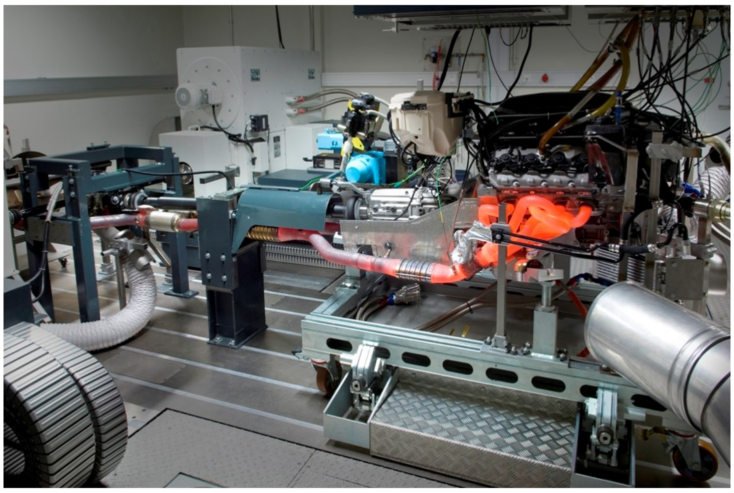


Figure 1.1: Engine testbed.

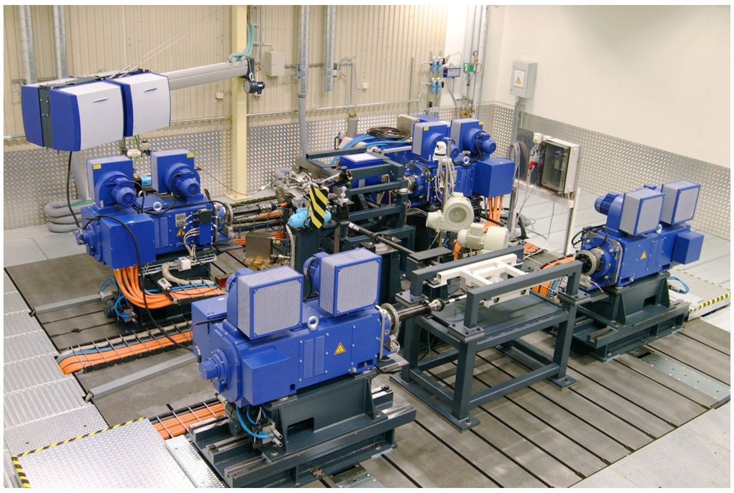


Figure 1.2: Powertrain testbed.

1.2 Outline of the Thesis

Chapter 2 covers a short overview of the testbed configuration, implemented control concept and the structure of the speed observer. Chapter 3 deals with the identification and tuning concept as well as the practical tests on individual automotive testbeds. In addition, the evaluations of the tests are also clearly presented in chapter 3. The subsequent chapter 4 describes a tool for creating test runs as well as some identification test runs which are necessary for the automatic tuning mechanism. The software implementation of the elaborated identification algorithm is described in chapter 5. This chapter also covers the handling of the identification software and describes the steps, which are necessary for the identification process at an automotive testbed.

2 Control Concept of the Automotive Testbeds

This chapter deals with a compact overview of the automotive testbeds from AVL that are relevant for this master thesis. Furthermore, it will explain the control concept, software architecture and the developed speed observer of these automotive testbeds.

2.1 Testbed Introduction

Basically, the automotive testbeds from AVL can be divided into two main groups with different configurations: There is the engine on the one hand and the powertrain testbed on the other hand. All these testbeds use one or more electrical machines to drive or load the unit under test. In this thesis and also at AVL, the electrical load machines or electrical driving machines of the testbeds are called *dyno* in short. The software package of AVL, which controls and monitors the dynos and the UUT, is called *EMCON*. The software name *EMCON* stands for **E**ngine **M**onitoring and **Con**trol.
At the testbed, the control software package *EMCON* needs a generic software which processes tasks that are not part of the controlling mechanism, but necessary for a control concept, which is working properly. This generic software is called *PUMA* (**P**rüfstands- **u**nd **M**esstechnik-**A**utomatisierungssystem) and it was also developed by AVL. Figure 2.1 shows a basic system for a better understanding of how the individual parts of a testbed interact with each other.

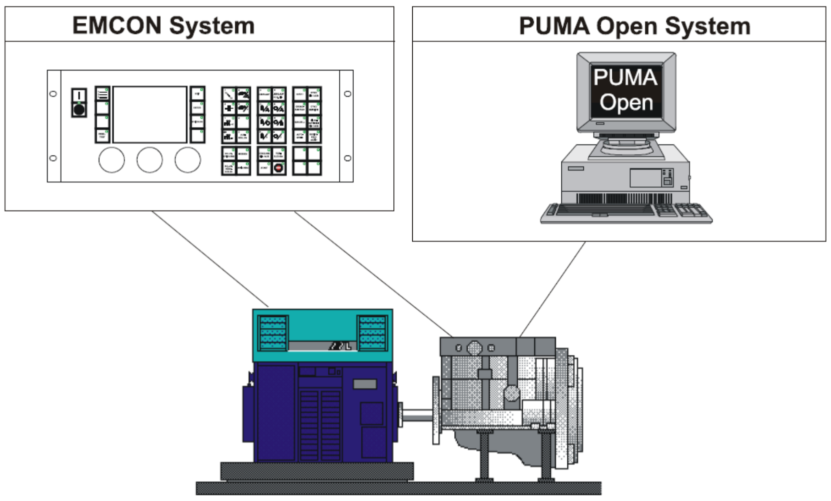


Figure 2.1: Testbed environment

The following figures 2.2 and 2.3 illustrate a schematic representation of an AVL engine and powertrain testbed. The engine testbed configuration has one dyno, which loads the combustion engine. In contrast, a powertrain testbed can have up to four individual dynos depending on the powertrain type. A testbed for testing a transmission has one dyno, a front- or back-wheel drive testbed needs two dynos and an all-wheel drive testbed, like in figure 2.3, has four dynos. However, independent of the testbed type, the dynos always have the task of simulating the environment for the unit under test.

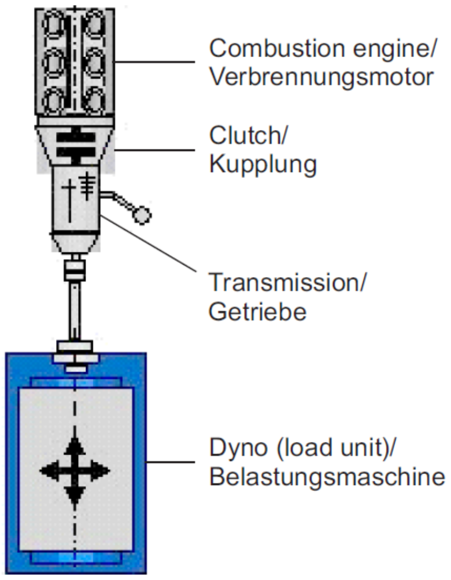


Figure 2.2: Engine testbed from AVL

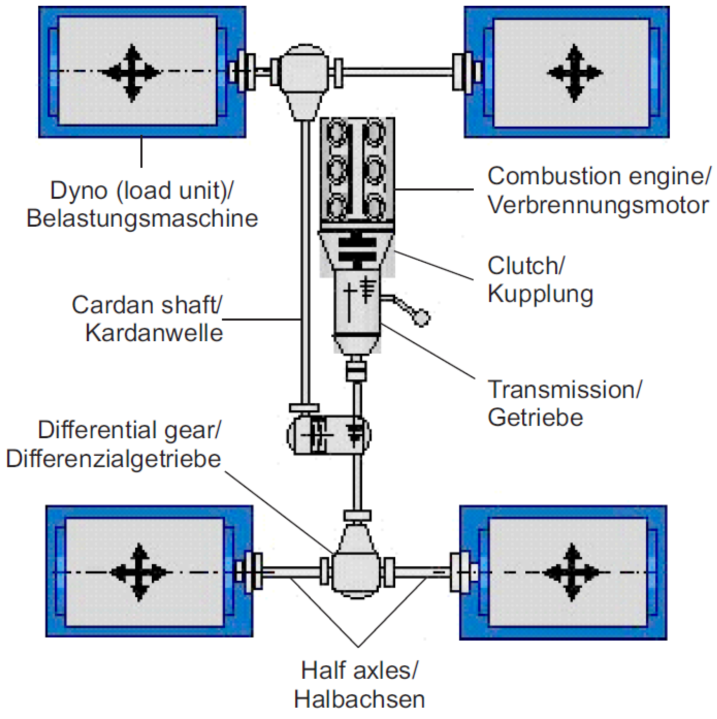


Figure 2.3: Powertrain testbed from AVL.

Currently, AVL offers a PUMA automation software package with a separated control software architecture. One part of the control software *EMCON* runs on the *PUMA* PC and another part is sourced out to an external control unit called *KIWI-Box*.

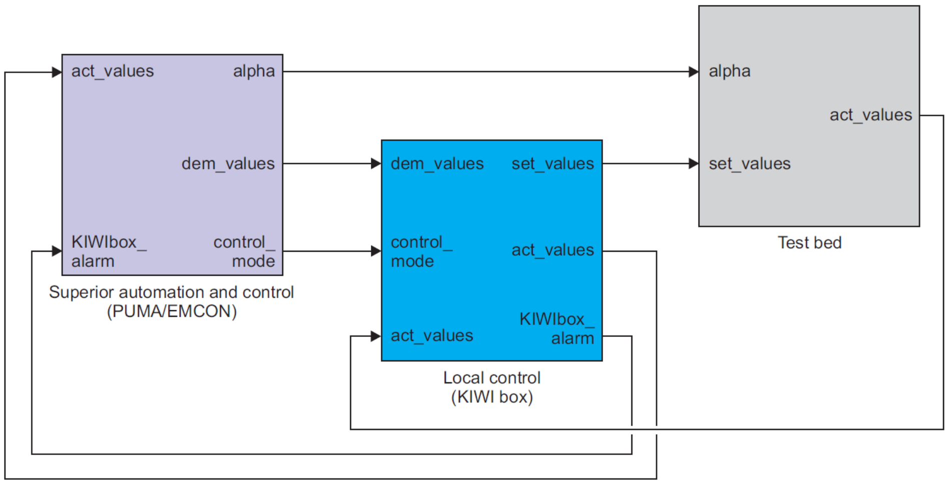


Figure 2.4: Distribution of control between PUMA/EMCON and KIWI-Box

The control architecture is illustrated in figure 2.4. The splitting of the control software *EMCON* is based on the different controlling speeds. The slower control part for the combustion engine and vehicle simulation is located in the *PUMA/EMCON* system. The dyno control (electric drive unit and load units), which represents the fast controlling part, is located in the *KIWI-Box*. A speed observer is assigned to each dyno of the testbed. It provides the speed signal for the speed controller and enables a correction of the torque set value (see figure 2.5). For this reason, the observer is responsible for the control behaviour of the entire automotive testbed. If the speed observer does not work properly, in case of wrong parameterisation, the testbed becomes instable and accordingly unusable. Therefore, it is important to tune the speed observer properly in order to get good controlling results and a well-working testbed. Figure 2.5 shows a schematic representation of the implemented control concept.

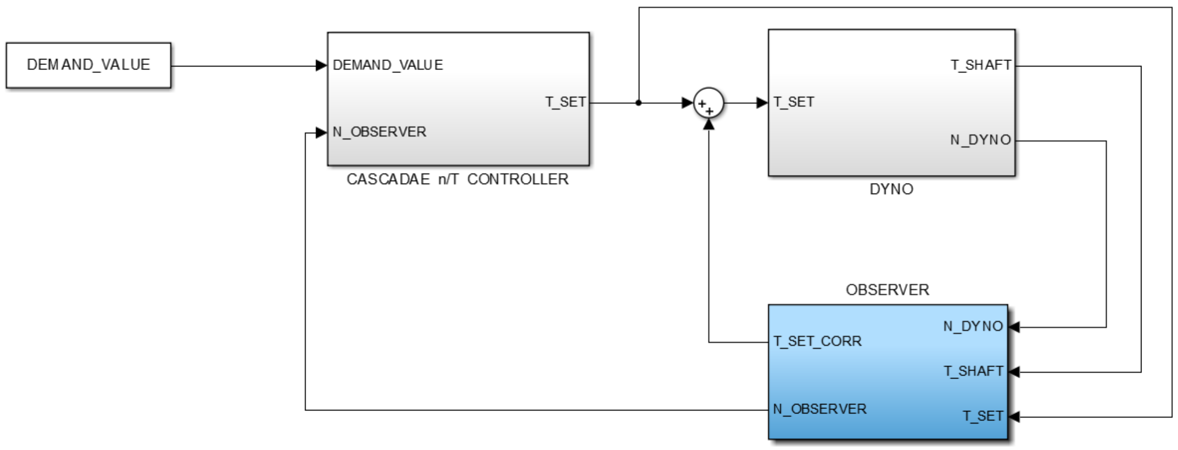


Figure 2.5: Schematic figure of the dyno speed control concept

2.2 Speed Observer

Generally, an observer is used for estimating hardly measurable or non-measurable system states. These observed system states can be used for controlling the system behaviour via a state controller. According to the definition of [6], an observer is an algorithm that estimates the system states with the help of the input values and the corresponding output values of the system. In contrast, the implemented speed observer returns the output value of the observed system. This output value is measurable and it is also used as an input for correcting the model inaccuracies. Thus, it is not implemented for getting a hardly measurable or non-measurable system state. It is rather deployed for filtering out measurement noise and other signal disturbances, which will be coupled into the control loop. That injection of occurring disturbances can lead to an unstable control loop. The dyno speed observer is used for avoiding these effects. It couples a continuous and noiseless speed signal back to the system controller. In this case, the implemented speed observer is not an observer in the usual sense, but it is rather used as an intelligent filter to remove noise and other side-effects from the dyno speed signal. However, the AVL-specific name of this component is “speed observer“, which is why it is also referred to as “speed observer“ in this master thesis.

2.2.1 Modelling an Electrical Engine

The dynos at an automotive testbed are three-phase motors with an inverter. The inverter converts a demand torque value into a corresponding voltage and current, and applies it to the electrical motor. Thus, a dyno model, like it is shown in figure 2.6, results in the following differential equation:

$$\Theta \frac{d\omega}{dt} = T - T_F \tag{2.1}$$

The formula symbol T represents the driving torque of the dyno and it is also the input quantity of the system. T_F symbolizes all occurring losses during the operation. These losses are not only friction losses but also machine-related losses, for example eddy current losses, heat losses and other unknown losses. The formula symbol Θ involves all occuring moments of inertia. Acceleration is symbolized by the time derivative of the speed ω in the equation 2.1.

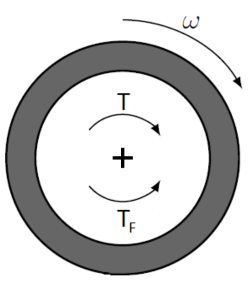


Figure 2.6: Simple mechanical model for electrical units

The equation 2.1 describes the essential behaviour of an electrical unit, but it does not contain some common basic technical effects. Distributed or networked systems, which are also available at the automotive testbed (see figure 2.4), always have to communicate between all system

parts. The transmission of the torque demand value from the speed controller to the frequency inverter requires some milli or micro seconds considered as communication delay time T_d. Neither communication nor building-up torque can happen in zero time. The speed of building-up torque (rise time t_r) is limited for physical reasons. The inverter needs time to convert the demand torque value into the corresponding voltage and current, and the inductance of the electrical machine opposes the change in current (self-inductance). Therefore, it takes time until the full demand torque is reached. These delay times can be summed up as dead time that must be considered for modelling the dyno.
Another physical effect are the speed-dependent losses. On higher revolution speed, higher losses T_F occur.
Considering the effects described above, the formula 2.1 for the dyno model s changed to the following:

$$\Theta \frac{d\omega}{dt} = T\left(t - T_d, t_r\right) - T_F\left(\omega\right) \tag{2.2}$$

2.2.2 Structure of the Dyno Speed Observer

As already mentioned, every dyno of the automotive testbed has its own speed observer. The dyno observer delivers a revolution speed signal based on the dyno model. This model describes the dyno inertia and the torque build-up dynamic, which consists of the delay and rise time of the dyno. The mathematical description of the dyno model is shown in equation 2.3.

$$\Theta \frac{d\omega}{dt} = T\left(t - T_d, t_r\right) - T_{shaft} - T_{corr} \tag{2.3}$$

The difference between formula 2.2 and the dyno model equation 2.3 is that the speed-dependent losses are not considered, but there are two additional torque values included, which have an influence on the currently active dyno torque: The shaft torque T_{shaft} on the one hand and the torque of the correction controller T_{corr} on the other hand. The torque T_{shaft} describes the measured torsional moment between the connection of the electrical unit and the unit under test. Depending on the operating state, the unit under test can drive or break the dyno. The shaft torque has an upper and a lower limit. These limits are always to be checked during the operation, because an overrun of these limits can lead to irreparable damages of the automotive testbed.
The other torque value T_{corr} represents the output of the correction controller, which should compensate the inaccuracies of the dyno model. These inaccuracies lead to a difference between the observed speed and the real revolution speed of the electrical unit. This correction controller revises the torque of the dyno model depending on the error between real and observed speed.
The output of the correction controller reaches the phase controller, which corrects the set torque of the real dyno. This additional phase controller corrects the phasing of the dyno shaft. This is important for powertrain testbeds in order to eliminate tensions in the transmission of the UUT. As discussed, every dyno has a speed observer, which provides the reference signal for the controller. The correction controller of an observer eliminates only the difference between measured and observed dyno speed, but neglects the actual shaft phase. Therefore, it is possible that two dynos, which are coupled together by a powertrain, have the same speed, but different shaft phases. Such a phase shift can actually be so strong that irreversible damages occur on the unit under test. The phase correction controller is used to avoid this scenario. This controller must generally be much slower than the correction controller of the observer. It works like an

outer controller of a cascade controlling structure. As a rule of thumb (see [12]), the integral action part *KiPhCorr* of the phase correction controller must be set smaller than the integral action part of the correction controller for the observer *Ki* by a factor of 10. For this reason, the parameter *KiPhCorr* is always one-tenth of the parameter *Ki* of the correction controller. The phase correction has no effect on an engine testbed and should not be used there. The controller can be set inactive, if the parameter *KiPhCorr* is put to zero. The figure 2.7 shows the structure of the speed observer.

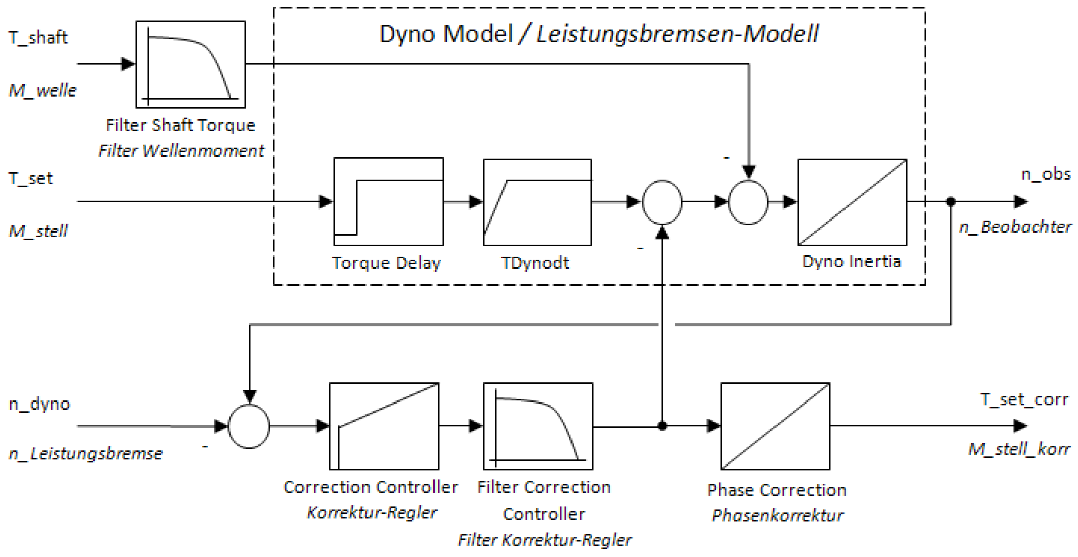


Figure 2.7: Dyno observer structure

The observer has many parameters, which can be divided in three parameter groups:

- model parameters

- controller parameters

- filter parameters

The following tables of parameter groups give an overview with a short description of all available as well as adjustable parameters of the speed observer.
The following parameters in table 2.1 are used to model the behaviour of the electrical units at the automotive testbed.

Parameter name	**Description**
`Inertia`	This parameter represents the total inertia of the electrical unit (*formula symbol* Θ). It is the sum of the individual dyno and shaft inertias.
`Torque Delay`	The *Torque Delay* parameter models the build-up delay of the current system. It is also known as the reaction time of a technical system. The delay time commonly consists of the communication time between actor and controller and the processing of the data in the arithmetical unit.
`TDynodt`	This property is used to indicate the torque rise time. It is physically impossible for the desired torque value to change infinitely rapidly. In this case the parameter specifies the time needed for arising the nominal torque from zero.

Table 2.1: Model parameter table.

Table 2.2 describes the parameters which are used for setting up the correction and the phase correction controller. The correction controller is a PI controller with additional look-up table parameters, which modify the proportional and integral value of the controller depending on actual dyno speed. These look-up tables are used because of the limited dyno speed sensor resolution. On low dyno speed it may happen that the sensor cannot register a speed variation during each sample time. Therefore, the sensor returns the same speed value over a period of time, after it measures the real, actual dyno speed. On slow revolutions, the error between observed and measured speed can be changed more abruptly than on higher speed values, because of this sensor characteristic. Thus, the look-up tables are established for a good control behaviour over the full-speed range. If the dyno speed goes below a defined value, the standard proportional and integral parameter will be reduced by a curve which is defined by these look-up table parameters.
The phase correction controller is an I controller. It also uses a look-up table for reducing the standard parameter on slow dyno speed like the correction controller.

As already mentioned above, a main task of the speed observer is to filter out measurement noise and other signal disturbances which are induced by the testbed sensors. The two measured quantities, which are needed for the speed observer, are the shaft torque T_{shaft} and the dyno speed N_{dyno} (see figure 2.7). The shaft torque quantity is directly filtered before the value has an influence of the calculated dyno model torque. The measured dyno speed value acts as an input for the correction controller. In this case, the output of the controller is filtered before it corrects the dyno model torque. The implemented filters have two parameters. These filter parameters are described in table 2.3.

Parameter name	**Description**
`Kp`	Proportional action part of the PI correction controller of the speed observer.
`Ki`	Integral action part of the PI correction controller of the speed observer. *Ki* is the reciprocal of the time constant of a PI controller's integral action part (unit 1/s).
`KiPhCorr`	Integral action part of the phase correction controller. It is also the reciprocal of the time constant of the integral action part (unit 1/s).
`KpCurve`	This parameter is used to reduce the *Kp* value of the PI correction controller at low speeds.
`KiCurve`	The property *KiCurve* is used to reduce the *Ki* value of the PI correction controller at low speeds.
`KiPhCorrCurve`	This is used to reduce the *Ki* value of the phase correction controller at low-revolution speeds.

Table 2.2: Controller parameter table.

Parameter name	**Description**
`Filter Type`	Determines the type of the filter. Each of the filters can be operated either as low-pass filter (Filter Type = 1) or as notch filter (Filter Type = 2).
`Filter Frequency`	This parameter defines the cut-off frequency of a low-pass filter or the trap frequency of a notch filter.

Table 2.3: Filter parameters table.

3 Parameter Identification

A main requirement of the observer parameter calculation is that the identification algorithm should operate quickly and as simply as possible. Furthermore, it should be independent from the automotive testbed configuration and dyno types. The type or rather the specification of a testbed dyno depends of the UUT. Because of the innumerable units under test, the range of the dyno types reaches from an asynchronous electrical machine with high torque performance to a powerful synchronous high-speed unit. Due to this requirement, the identification algorithm should be kept as general as possible.
Since working on real testbeds was not always possible for this thesis, the identification concept would be completely elaborated and tested by a simulation model in Simulink®. Only the finished identification method with the corresponding test runs were carried out on individual automotive testbeds.

3.1 Model Parameters

The central parameters of the speed observer are the model parameters, which describe the behaviour of a testbed dyno. The mathematical modelling of the dyno is expressed by the equation 2.3 in chapter 2. This equation consists of several formula variables that need to be determined. The method to identify these unknown model parameters is to execute special test runs without connecting any unit under test. This means that the dyno is decoupled from the engine or powertrain during the test run. Therefore, it is possible to record the behaviour of the dyno without any influence of other system components. The calculation of the unknown parameters is based on the recorded demand torque and output speed signal of these defined test runs.

3.1.1 Operational Losses Calculation

The operational losses of an electrical machine depend on the current revolution speed. A dyno records higher losses on a higher speed. The interested reader is referred to [15] for further details. Because of this, it is important to define the speed and also the torque point for the parameter identification.
The electrical units of the testbed are selected in such a way that the average operating scenario is in the range of the nominal speed and nominal torque. Therefore, the whole identification concept as well as test runs operate at the nominal point of the given electrical unit.

Based on the model equation 2.2, it is obvious that the dyno-based losses are easily measurable if the acceleration is set to zero $\left(\frac{d\omega}{dt} = 0\right)$. To achieve this situation, the electrical machine must drive a constant speed value. If a constant speed is reached, the equation changes to the following structure:

$$0 = T\left(t - T_d, t_r\right) - T_F\left(\omega\right) \tag{3.1a}$$

$$T\left(t - T_d, t_r\right) = T_F\left(\omega\right) \tag{3.1b}$$

$$T = T_F\left(\omega\right) \tag{3.2}$$

In the steady state of the dyno (constant speed), the delay time as well as the torque rise time have no influence. The model equation simplifies to equation 3.2. At this stationary condition, the controller needs to set up the torque value, which compensates the losses at the current revolution speed. Therefore, the controller output T represents the machine-based losses at the given constant speed value.
Generally, the machine-based losses are depending on the current revolution speed of the testbed dyno. This phenomenon can be easily demonstrated by a coast down experiment as shown in [8]. In this experiment, the dyno speed has to be controlled to a high speed level first and after that, the input torque value should be set to zero to start the coast down process. Therefore, the dyno stops only by the friction torques. This coast down experiment was performed at a testbed dyno and is illustrated in figure 3.1 (In every diagram of this master thesis, the dyno speed ω (rad/s) is represented in rotation per minute (rpm) with the formula symbol N.).

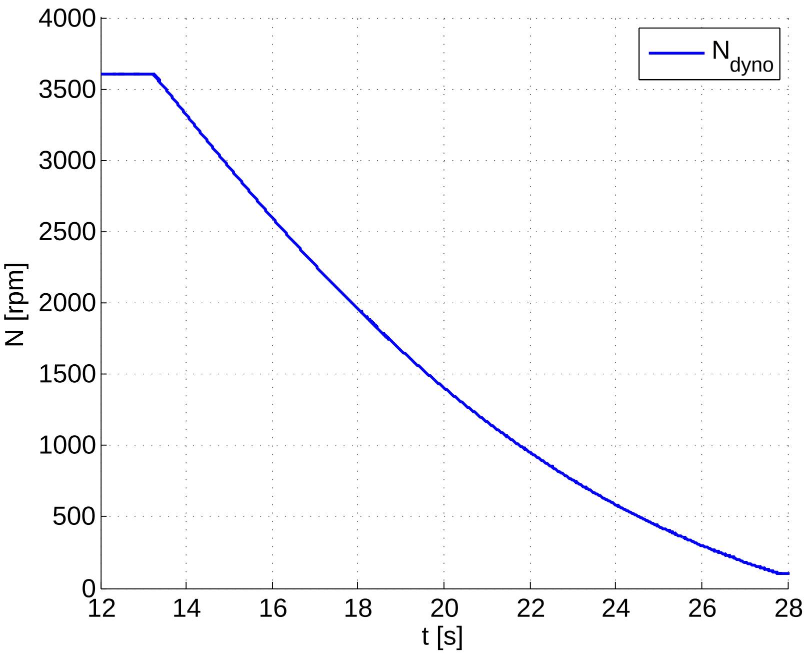


Figure 3.1: Coast down experiment of a testbed dyno.

The resulting speed characteristic has a non-linear progression, because of the speed-dependent losses. Thus, the following friction model results:

$$T_F\left(\omega\right) = T_C\ sign(\omega) + T_V\ \omega \tag{3.3}$$

The formula symbol T_C specifies a positive parameter for the dry (Coulomb) friction and T_V represents a positive parameter for the viscous friction.
The friction value T_F is not a model parameter (see table 2.1) and should be compensated during the operation by the correction controller of the speed observer. The only reason for identifying T_F is for better determination of the model inertia parameter. In this case, the friction model (see equation 3.3) is simplified to the following for a simpler identification of $T_F\left(\omega\right)$.

$$T_F\left(\omega\right) = T_C\ sign(\omega) \tag{3.4}$$

The value T_F is calculated by the operation losses around the nominal speed. This means that the occurring losses near the nominal speed will be averaged to one constant friction value. The test run for calculating the friction value controls the testbed dynos to their nominal speed and persists there for a few seconds. The same is carried out for 90 % and 110 % of the nominal speed. The three received speed dependent losses at these speed values are averaged to one dyno loss quantity T_F.

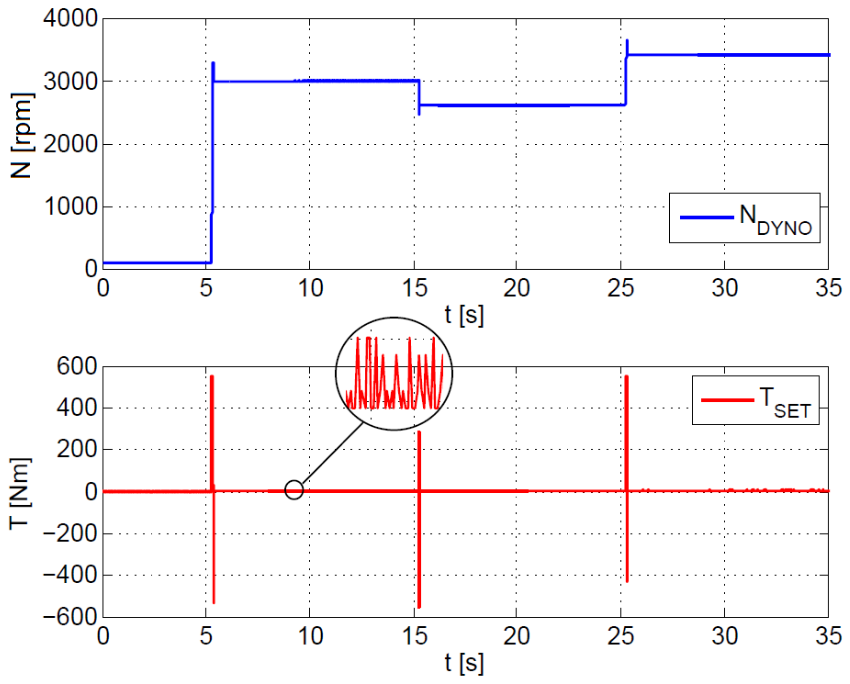


Figure 3.2: Test run for identification of the dyno based losses.

Figure 3.2 shows the described test run for the identification of the dyno losses. The upper sub plot represents the measured dyno speed and the sub plot below shows the applied torque from the speed controller. The figure 3.2 also demonstrates that the measured signals are always fraught with measurement noise, which leads to a distorted measurement result, if only one single torque value is selected to represent the losses of the dyno. To compensate this relative measurement error, the average value of a torque range at the steady state is calculated the same way as the machine-based operating losses.

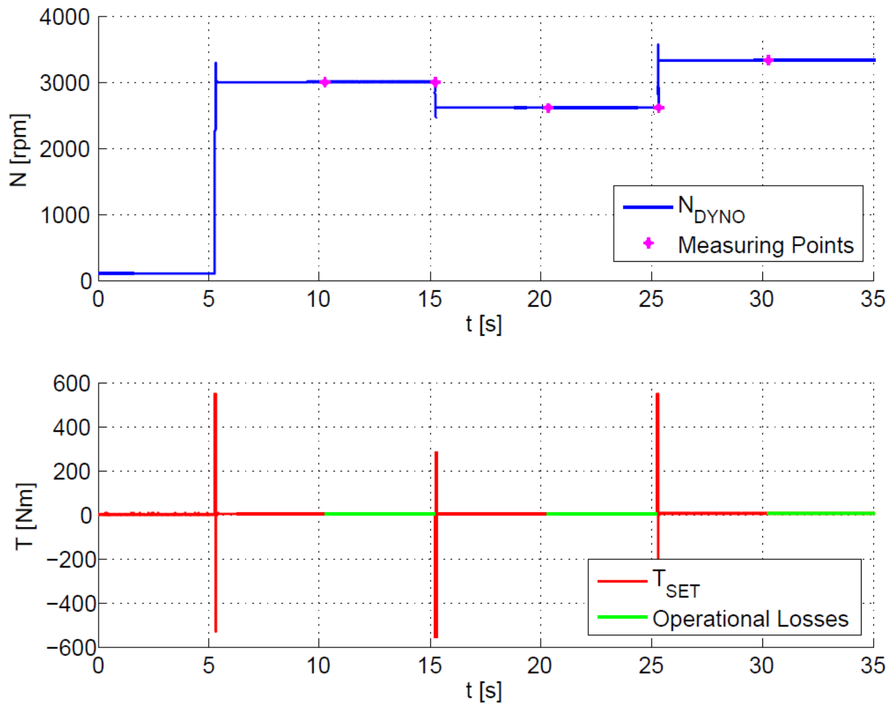


Figure 3.3: Calculation of the dyno's operational losses.

Figure 3.3 shows a graphical representation of the identification mechanism from the operational losses of the dyno. The magenta coloured dots are the detected start and end points of the steady states from each demand speed and the green lines represent the area for calculating the loss value of the dyno. Therefore, all green T_{SET} values of each demand speed are averaged to one friction quantity, which represents T_F of the corresponding speed value. Afterwards, these three determined friction values are averaged to one T_F value, which represents the operational loss of the testbed dyno and will be used for identifying the model inertia (see chapter 3.1.2).
This test run was performed on two different automotive testbeds and thus on two different dyno types. One of these testbeds was a combustion engine testbed with a synchronous machine. This synchronous three-phase motor has a nominal speed of 3000 rpm and a nominal torque of 500 Nm. The other testbed was a powertrain testbed with two asynchronous dynos. These three-phase dynos have a nominal speed of 650 rpm and a nominal torque of 3000 Nm. Table 3.1 and 3.2 illustrate the calculated results of the test run at both automotive testbeds.

Speed [rpm]	T_F [Nm]
2700	4.1310
3000	4.5214
3300	4.7124
Mean Value	4.4549

Table 3.1: Test run results of the synchronous motor.

Speed [rpm]	T_{F_Dyno1} [Nm]	T_{F_Dyno2} [Nm]
585	20.0901	14.0880
650	16.0494	10.8272
715	12.5326	8.5482
Mean Value	16.224	11.1545

Table 3.2: Test run results of the asynchronous motor of the powertrain testbed.

3.1.2 Inertia Calculation

Moment of inertia is a measure of an object's resistance to changes in a rotational direction [3]. The inertia of a body, or in this case of a dyno, depends on the distribution of the dyno mass with respect to the axis of rotation. Therefore, the inertia represents the main dynamic behaviour of the dyno model and even small deviations implicate unequal behaviour between dyno model and a real electrical engine.
By reshaping the dyno model equation 2.2 and the previously calculated friction parameter T_F (see chapter 3.1.1), the inertia of the dyno can be calculated by another identification test run. During this test run, the input torque must be set to a constant value for a certain period of time. As a result of this constant demand torque, the delay time and the torque rise time can be neglected. Thus, the dyno model equation 2.2 for calculating the inertia can be simplified as follows:

$$\Theta = \frac{T - T_F(\omega)}{\frac{d\omega}{dt}} \tag{3.5}$$

Equation 3.5 demonstrates the possibility of calculating the dyno inertia by the previously measured losses $T_F(\omega)$, the constant applied input torque T and the gradient of the dyno speed $\frac{d\omega}{dt}$.
There are two ways for setting the input dyno torque T to a constant value. One way, like in [8], is to set the input value to zero for obtaining the dyno characterisation. This is shown in equation 3.6 and figure 3.1.

$$\Theta = \frac{-T_F(\omega)}{\frac{d\omega}{dt}} \tag{3.6}$$

According to this coast down operation, the inertia is the ratio between the machine-based losses and the falling gradient of the measured speed value.
The second way to find out the inertia, is the opposite procedure. Initially, the dyno has to be controlled to a low level speed value, and afterwards a torque jump with a steady value has to be applied. By determining the rising gradient of the speed, it is also possible to calculate the inertia of the given dyno by solving the equation 3.5. The advantage of the identification of the dyno inertia with the coast down test is that this experiment is very simple to perform. The coast down process could be easily realized with all electrical units and without any major efforts. However, the disadvantage is the temporal component of this identification mechanism. The time duration of the coast down process depends on the dyno type. A permanently excited synchronous machine needs only a few seconds for a coast down, whereas a well-mounted asynchronous machine with a high inertia might need more than half an hour until it stops. In contrast, the disadvantage of the torque jump experiment is the realization of the test run. In order to not run the dyno for too long at maximum speed or even create a speed overshoot, the duration of the torque jump should be limited. This can easily be achieved by the automation software PUMA, which provides a comfortable environment to define and execute such test runs. The duration of this identification mechanism is, on the other hand, an advantage for this practice. It does not matter, which kind of dyno is used, it always takes only a few seconds.
Another huge advantage is that this inertia identification test run can also be used for determining the other two model parameters (Torque Delay and TDynodt). This calculation will be explained in the following chapters.

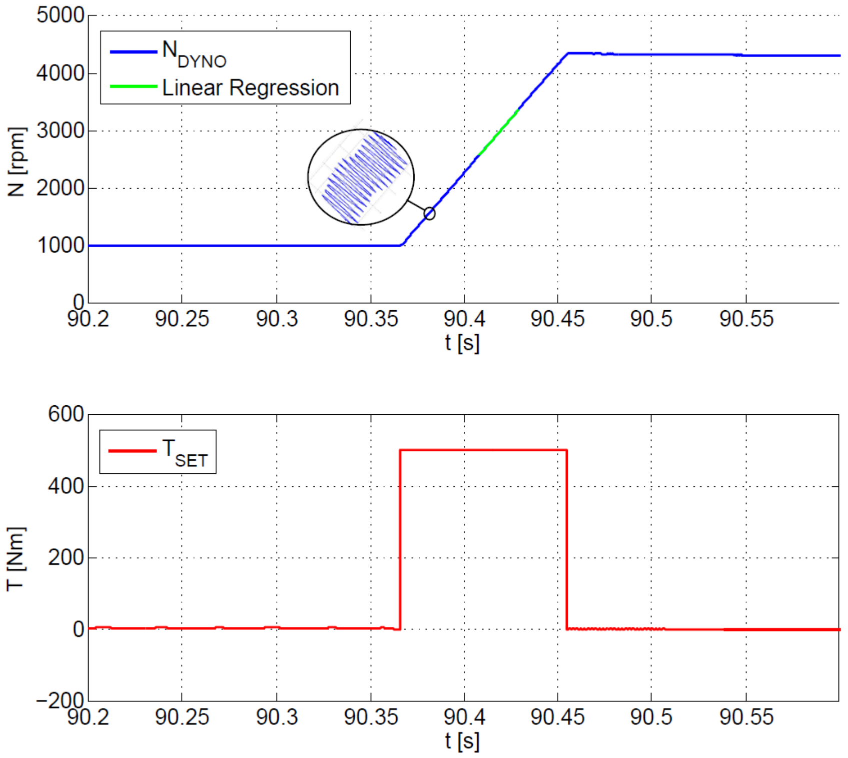


Figure 3.4: Test run for determining the dyno inertia.

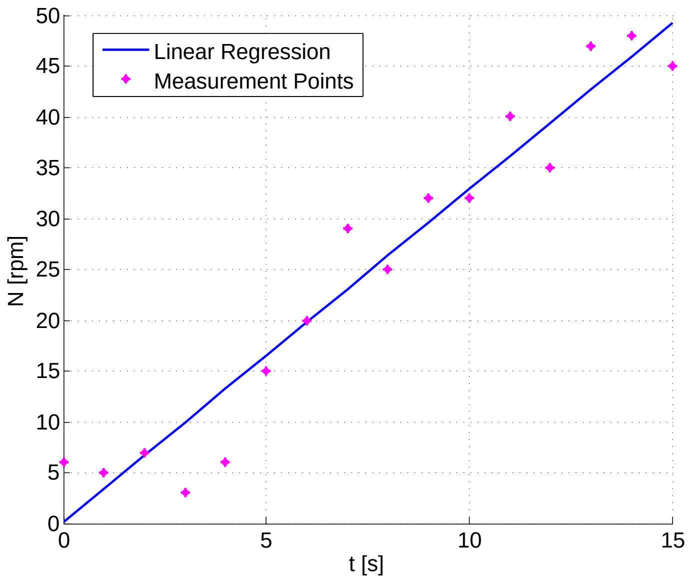


Figure 3.5: Simple example for a linear regression.

Figure 3.4 shows the recorded torque jump test run on the combustion engine testbed for the dyno inertia identification. In the first part of this test run, the dyno is controlled to 1000 rpm and the nominal torque of 500 Nm is applied until 90 % of the maximum dyno speed is reached. Because of the signal noise, it is not expedient to calculate the gradient at one point of the speed signal (see figure 3.4). Therefore, the gradient of the range between ±10 % of the nominal dyno

speed (here 3000 rpm) is calculated by linear regression. This is the reason why the operating loss is also calculated in the same speed range (see chapter 3.1.1). This identification section (nominal speed ±10 %) is considered to be linear for detecting the dyno model parameters.
Linear regression generally attempts to model the relation between two variables by fitting a linear equation. One variable is considered to be an explanatory variable X and the other as a so-called dependent variable Y. In this case, it should produce a linear relationship between the time values X and the speed values Y. The following figure 3.5 illustrates a very simple example for a linear regression of some randomly selected speed points.
The mathematical derivation of this linear regression example looks as follows:

$$\begin{aligned} y_1 &\approx k\ x_1 + d \\ y_2 &\approx k\ x_2 + d \\ &\vdots \\ y_n &\approx k\ x_n + d \end{aligned}$$

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}} \approx \underbrace{\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} k \\ d \end{bmatrix}}_{\mathbf{p}}$$

At the representative example above, it is assumed that a linear relation exists between n measured speed points. An equation of a line is set up for every measurement point. The values $y_1 \dots y_n$ illustrate the recorded n speed values, $x_1 \dots x_n$ the corresponding time stamps, k represents the slope and d the offset of the requested fitting line, which describes the linear progress of the measured speed. Because of the given overdetermined system of equations, there is no vector **p** which fulfil all equations. Therefore, approximate signs are used instead of equal signs in those equations above.
According to a proposal by Carl Friedrich Gauß, a German mathematician, a solution for vector **p**, which minimizes the sum of squared errors between points and fitting line, should be found. This solution is obtained if the given overdetermined system of equations is transformed to equation 3.7.

$$\mathbf{p} = \mathbf{M}^{+}\mathbf{y} \tag{3.7}$$

The matrix $\mathbf{M}^{+}$ in equation 3.7 represents the pseudoinverse of matrix $\mathbf{M}$. A pseudoinverse is a special type of an inverse matrix, which can be used for singular and non-square matrices. It is commonly used for calculating least square solutions of linear systems of equations, like in the given example above. The interested reader is referred to [2] and [6] for further details on linear regression and pseudoinverse.
This described line fitting process is performed for calculation the slop ($k = \frac{d\omega}{dt}$) of the recorded speed signal between 90 % and 110 % of the nominal dyno speed. On the basis of this slop calculation, the dyno inertia can be identified through equation 3.5.

The torque jump test run for calculating the inertia was performed at the engine and powertrain testbed just in the same way as the identification of the operational losses. Tables 3.3 and 3.4 represent the results of the calculated inertia as well as the inertia specified by the manufacturer.

	Specified Inertia [kgm^2]	Inertia (TJ) [kgm^2]	Inertia (CD) [kgm^2]
Engine Dyno	0.1090	0.1257	0.1184

Table 3.3: Inertia specified by the manufacturer and the calculated inertia of the engine testbed dyno.

	Specified Inertia [kgm^2]	Inertia (TJ) [kgm^2]
Dyno Front Left	4.0	4.0934
Dyno Front Right	4.0	4.0852

Table 3.4: Inertia specified by the manufacturer and the calculated inertia of the powertrain testbed dynos.

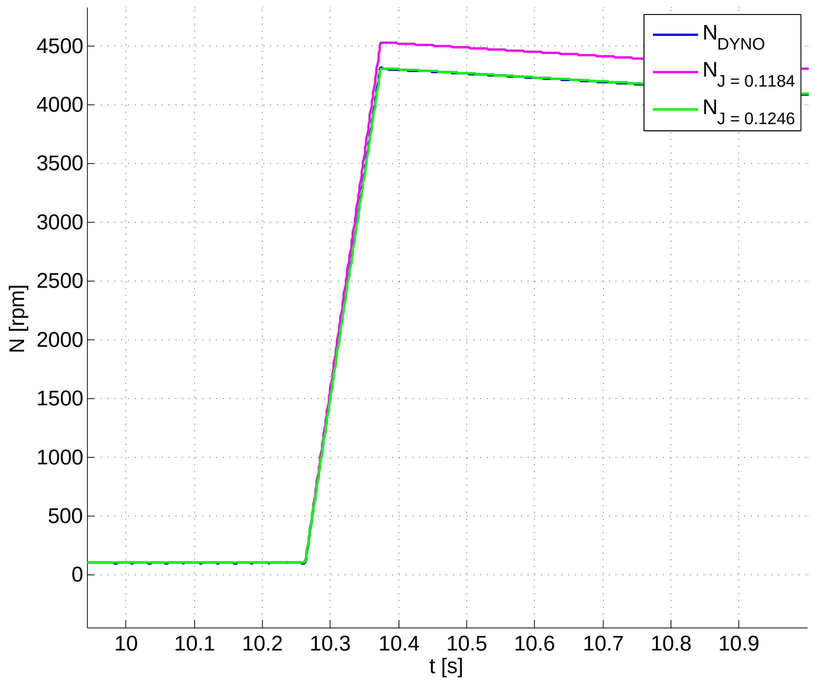


Figure 3.6: Comparison between real and simulated dyno speed behaviour.

The abbreviation *TJ* of the tables 3.3 and 3.4 stand for *Torque Jump* and represents the calculated dyno inertia by the torque jump test run. The abbreviation *CD* of the tables 3.3 stands for *Coast Down* and illustrates the calculated dyno inertia in the coast down test run.
From both tables above, it can be easily perceived that the identified inertia is always higher

than the inertia specified by the manufacturer. This was expected, because every testbed dyno has an additional flange for determining the dyno speed and the appearing torque at the shaft. Furthermore, the coast down experiment was also performed at the engine testbed for calculating the inertia. The coast down inertia is a little bit lower than the inertia of the torque jump test run. Figure 3.6 shows a validation of the identified inertia from the combustion engine testbed with the aid of a Simulink® model. This compared presentation indicates, that the dyno model with the torque jump inertia conforms better with the real dyno behaviour than the model with the inertia which is identified by the coast down experiment. The reason for that is that the torque jump inertia is derived from an operation process of the dyno, which represents the dynamic behaviour in a better way. Therefore, it may be assumed that some physical effects, which have an influence on the dyno inertia, occur during the operation such as eddy current losses.
In addition, figure 3.6 shows that already a small deviation between real and identified inertia generates a wrong dyno model speed behaviour.

3.1.3 Delay Time Calculation

Time delays are common in many technical systems. They are often caused by communication or measurement lags, delayed sensing and other influences. The delay time describes the time between applying an input signal and the resulting reaction of the system output. On controlled systems, time delays can cause instabilities and they affect the control behaviour adversely. For control engineering, it is important to know the exact delay time of the system for the design of the controller.

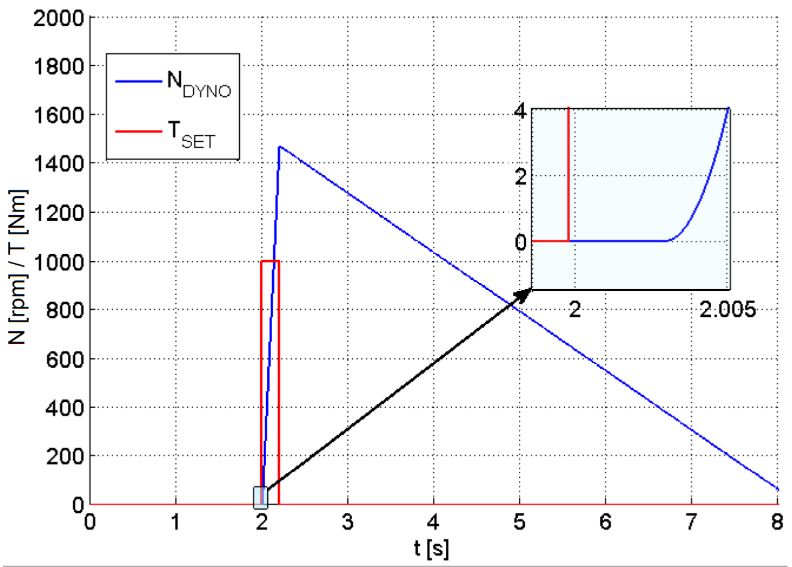


Figure 3.7: Delay time illustration by a Simulink® dyno model.

The delay, or so-called dead time, of a system is usually diagnosed by measuring the output signal during abrupt changes of the input. Such a measuring process is carried out with the torque jump test run for calculating the dyno inertia (see chapter 3.1.2). Therefore, this test run can also be used for determining the dyno delay time, which saves identification time, as no additional test run is required. Figure 3.7 demonstrates the effect and identification of the system delay time by the torque jump test run with a Simulink® model. The dead time is easily

recognizable in a simulation, as it is represented above in figure 3.7. Due to the noise, the exact determination of the delay time by real measurement data from an automotive testbed presents a more difficult task.
Figure 3.8 represents the recorded speed and set torque signal of a torque jump test run. It shows that the measured speed always oscillates around the demand-value. The first speed value change after the torque jump is caused by the noise and is not the reaction of a new input value. Before the impress of the torque jump occurs, the dyno is controlled at a constant speed value. This constant section is used for calculating the oscillation zone of the speed signal. The upper bound of this zone represents the threshold, which must be reached to signalise an output reaction triggered by the input.
The table 3.5 and 3.6 below illustrate the delay times of the dyno calculated by the torque jump test run method. As these tables show, the dead time of the dyno controlling system decreases when the torque jump value is increased. This effect definitely appears at both testbed systems. Furthermore, the torque jump test run was carried out by two different constant start speed values at the engine testbed, which also leads to different delay times.

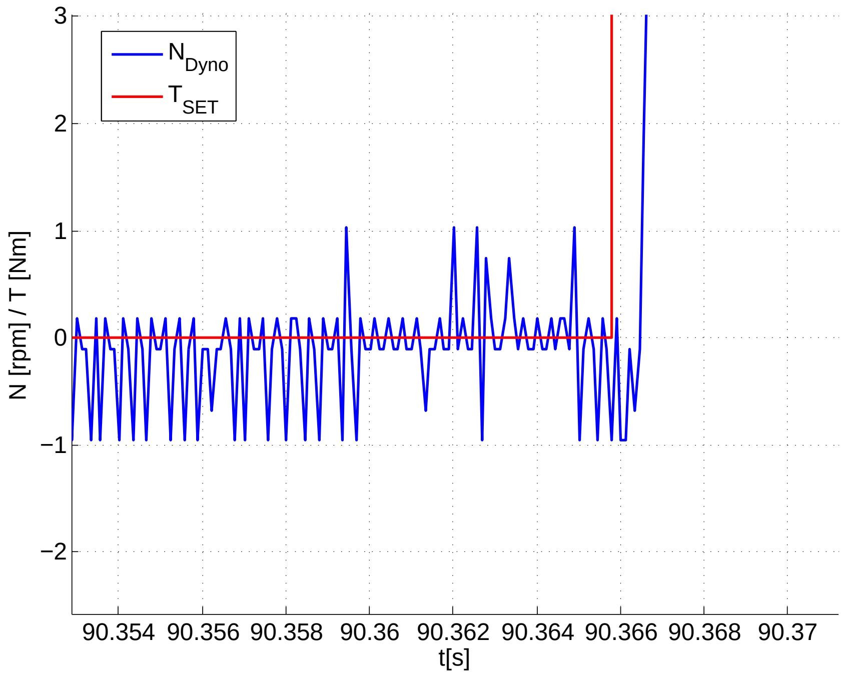


Figure 3.8: Delay time representation by real measurement data.

The column *Delay Time [ms] (100 rpm)* of table 3.5 represents the identified delay time with a start speed of 100 rpm and the other column, *Delay Time [ms] (1000 rpm)*, with 1000 rpm. The unequal delay times by the same torque jump value are caused by the limited resolution of the dyno speed sensor. The resolution of the dyno's build-in sensor for 100 rpm was too low for determining a variation of the speed value in every sample. Therefore, an arbitrary measurement error influences the identification and increases the delay time. This measurement error should be avoided for the identification of the time delay. In this case, it is important that the start speed of the torque jump test run is high enough to ensure that this technical sensor limitation does not apply. By the commonly used speed sensors of the AVL dynos, a start speed of about 500 rpm is sufficient for eliminating this measurement error. It should be noted, however, that the corresponding commissioning engineer is responsible for a correct choice of this start speed test run parameter.

Torque [Nm]	Delay Time [ms] (100 rpm)	Delay Time [ms] (1000 rpm)
50	3.3	1.9
100	2.9	1.2
300	1.2	0.8
500	1.3	0.8

Table 3.5: Identified delay time at the engine testbed

Torque [Nm]	Delay Time Dyno 1 [ms]	Delay Time Dyno 2 [ms]
500	6.2	6.6
1000	4.1	4.3
2000	3.0	3.0
3000	2.1	2.0

Table 3.6: Identified delay time at the powertrain testbed

The decreasing of the delay time by increasing the torque jump value is caused by a physical reason and the way of calculating it. As already mentioned, the recorded speed value must be higher than a defined threshold for detecting an output reaction depending on the applied input. If the torque jump value is too low, it may happen that the dyno speed exceeds that threshold after some samples, although the full demand torque is applied. This essential effect is shown below in figure 3.9. It represents the comparison of the recorded torque jump test run speed signals from the engine testbed by individual torque values. In this plot, the demand value is applied at time zero for every speed signal. It can be easily recognized, that the speed of the red and green curve increases slower than the other plotted speed signals. In this case, the higher detected delay time value is a result of the slow speed increase, which is caused by the low demand torque. Figure 3.10 illustrates the same dyno behaviour comparison on the basis of the dyno Simulink® model. The missing torque rise time parameter value for the simulation model (see figure 3.10) is taken from chapter 3.1.4, which describes the identification of the last model parameter. This effect can also be shown by that simulation plot. For this reason, not only the start speed value is important for a good delay time identification, but also the choice of the torque jump value is an essential part for a good delay time detection.

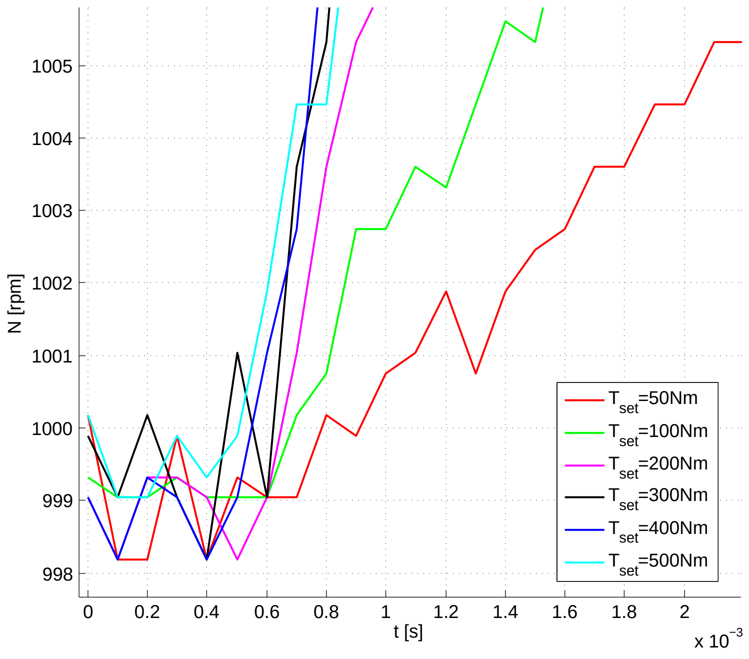


Figure 3.9: Measured torque jump test runs by individual torque values.

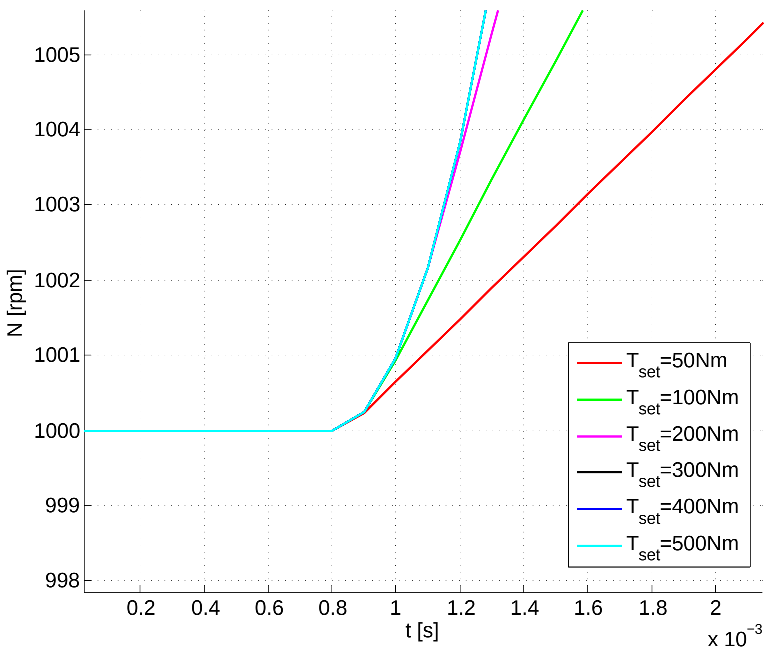


Figure 3.10: Simulated torque jump test runs by individual torque values.

chap:

3.1.4 Torque Rise Time Calculation

The last observer parameter of the dyno model is the torque rise time (*TDynodt*). This model parameter defines the torque build-up time from zero to the nominal torque of the given dyno and it is declared in milliseconds. It specifies the maximum available electrical torque change of the testbed machine. This parameter is therefore realized by a rate limiter block in the Simulink® model of the testbed dyno.
The rate limitation of the torque disposes that the applying input can only change in a continuous way on an abrupt input torque step. As a consequence, the waveform of the speed signal becomes quadratic until the demand value of the torque jump is reached. Therefore, it seems obvious to calculate the torque rise time from this quadratic path of the speed signal by fitting a polynomial curve. The gradient of the fitting curve presents the rate limit and thus the rise time parameter. This part of the speed characteristics is shown in figure 3.11. It appears that the torque jump test run is also useful for identifying the build-up torque parameter. However, the measured speed signal, which is shown in figure 3.12, represents, that the quadratic path is difficult to detect. In this case, the fitting curve identification mechanism works unreliably and is not suitable for the rise time identification. Instead, the Simulink® dyno model is consulted for calculating this model parameter.
At this time of the model parameter identification, all parameters except the rise time parameter are identified and available. Due to this, it is possible to identify this unknown parameter by a simple optimization process. The aim of the optimization problem is to minimize the error between the measured and the simulated speed signal($e_N = N_{Dyno} - N_{Simu}$). The Simulink® model is set up by the determined parameters and a low start torque rise time, e.g 100 ms. The input signal of the model is the recorded dyno controller torque signal of the torque jump test run. After the simulation, the error between the measured and received simulated speed signals is calculated. This process is repeated until the error reaches a minimum due to modifying the rise time. The figure 3.13 below illustrates the measured dyno as well as two simulated speed signals with different rise times (green = 20 ms, magenta = 10 ms). On the basis of this figure it is easily discernible, that the error between measured and simulated signal decreases at a higher torque build-up time.
MATLAB® already provides many different algorithms for the kind of optimization problem, as it is described in the following chapter.

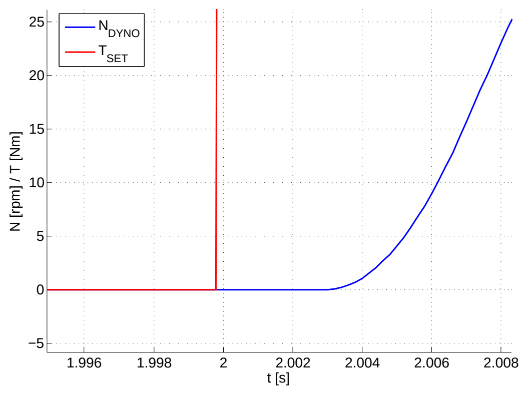


Figure 3.11: Simulated speed signal of the dyno model.

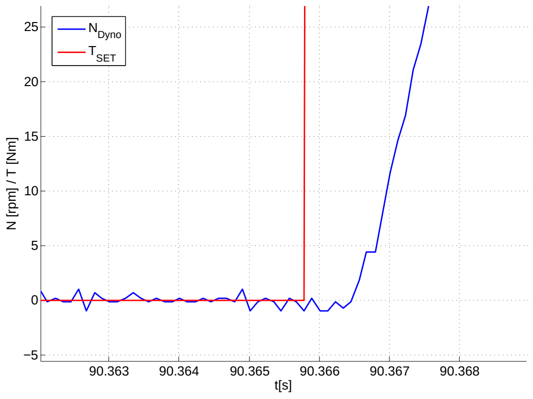


Figure 3.12: Measured speed signal of the engine testbed dyno.

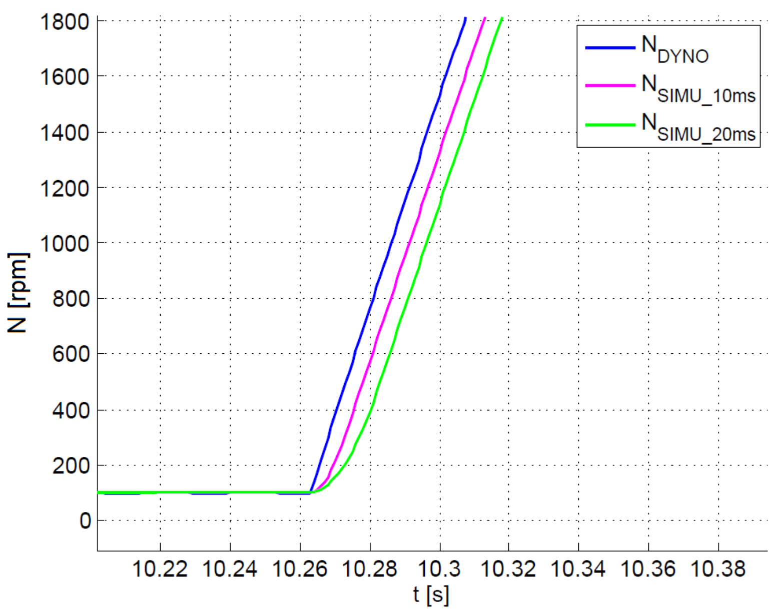


Figure 3.13: Comparison between real dyno speed and simulated speed signals with rise time parameter (TDynodt) variation.

3.1.4.1 Optimization with MATLAB®

Generally, optimization deals with the best selection of elements from a set of available alternatives [4]. In the most common cases, an optimization problem consists of maximizing or minimizing a given function by varying one or more function variables. The mathematical expression of an optimization problem is illustrated with the equations 3.8a and 3.8b. The function $f(x)$ represents the so-called objective function, Z constitutes the region of the optimization and x the modifying variable. The usual optimization algorithms are basically designed to find the minimum of the objective function, because every maximizing problem can easily convert to a minimizing optimization, which is shown with the equation 3.8b.

$$\min_{x \in Z} f(x) \tag{3.8a}$$

$$\max_{x \in Z} f(x) = \min_{x \in Z} -f(x) \tag{3.8b}$$

The optimization problems can be divided categorically into two issues: The linear optimization on the one hand and the non-linear optimization on the other..
Linear optimization strives to find a technique for the optimization of a linear objective function with potential additional linear equality or inequality constraints. Algorithms exist for linear optimization problems, which can determine the optimum or the unsolvability of the given problem in a finite number of steps.
Nonlinear optimization is the process of solving a nonlinear objective function over a defined set of unknown real variables with potentially linear equality or inequality constraints. In contrast to the linear optimization problems, no solving algorithms exist for nonlinear problem, which always returns the global optimum of the given problem. Therefore, it is sometimes only possible to calculate a local minimum/maximum of the objective function. But often this fact does not matter, because the common problems can be restricted by specified constraints in such a

way, that the local minimum/maximum becomes a global minimum/maximum of the defined region.

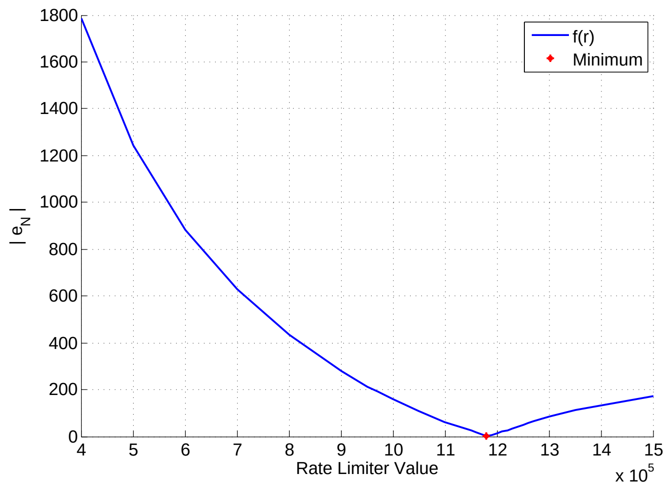


Figure 3.14: Objective function of the torque rise time parameter from the dyno of the engine testbed.

Figure 3.14 shows the final part of the objective function ($f(r) = N_{Dyno} - N_{Simu}(r)$) from the torque rise time optimization. The previously identified parameters of the engine testbed dyno are used for calculating this optimization objective function $f(r)$. The rate limiter value is manually increased linearly (x-axis) and the received absolute error between measured and simulated speed is plotted over the y-axis. It becomes clear immediately that the objective function has non-linear characteristics. In addition, the lower and upper bound of the optimization section can be found by a simple consideration. They represent the extreme cases of the rise time parameter.

The lower bound of the rate limiter is zero. This limit value means that the dyno inverter can not build up the demand torque in finite time.

The upper bound and thus the other extreme case is the torque rate value in which the torque rises from zero to the nominal value in only one sample time. For example, at a sample time of 1 ms and a nominal torque of 500 Nm, the upper slew rate limiter value is 500000 $\frac{Nm}{s}$. The global minimum of the absolute error has to be between these both bounds. Furthermore, it can generally be argued that the minimum is always closer to the upper than to the lower bound, because the aim of the dyno inverter controller is to reach the demand value in a reasonable time and manner. In that case, only the final part of the objective function will be interesting and it always looks like the illustrated function which is shown in figure 3.14.

A shift of the minimum to the left signifies that the rate limiter value decreases and a right shift represents an increasing limiter value. An up and down shift of the minimum point of the objective function provides information about the identification of the other model parameters. A high minimum value indicates that the identified inertia and delay time do not match the real dyno parameters well. Due to this, the objective function minimum also gives feedback on the parameter identification quality.

Based on this gained knowledge, various non-linear optimization functions from the MATLAB® *optimization toolbox* were tested with different settings under the same conditions. The performance or rather the runtime was determined for every constellation and it was the deciding argument for the algorithm selection. The following sections provide the description of the tested MATLAB® optimization function as well as the performance evaluation. The optimization was executed with the measured data of the engine testbed. The data were recorded with 4kHz and the nominal torque of the dyno was 500 Nm. Therefore, the lower (x1) and upper (x2) bound for the algorithms are:

$$\mathrm{x1} = 0\ \frac{Nm}{s}$$
$$\mathrm{x2} = 500\ \mathrm{Nm} \cdot 4000\ \mathrm{Hz} = 2000000\ \frac{Nm}{s}$$

fminsearch

This function belongs to the non-linear optimization algorithm and solves unconstrained minimization problems. It uses the *Nelder-Mead* or downhill simplex algorithm, which finds the function minimum by comparing the function values. Generally, this algorithm does not converge very fast but it is simple and robust.

SYNTAX: *x = fminsearch(fun,x0)*

fun ··· Represents the function to be minimized
x0 ··· Starting point point of the optimization algorithm

The parameter *fun* is a MATLAB® function handle, which returns the error between the measured and simulated speed signal. The optimizing rate limiter value acts as the unique input parameter of this function. The parameter *x0* defines the start slew rate with which the optimization should start. The table 3.7 shows the runtime as well as the optimum rate limiter value at different starting points *x0*. The runtime value was identified by the stopwatch timer (command: *tic toc*) of MATLAB®.

$x0$ [Nm/s]	Rate Limiter Value [Nm/s]	Runtime [s]
x1	1193911	24.3
x2	1193911	13.1
x2/2	1193911	12.1

Table 3.7: Evaluation of the *fminsearch* function

As table 3.7 above suggests, the *fminsearch* function always identifies the same rate limiter optimum value, independent from the starting point. The starting point *x0* only has an effect on the algorithm runtime. The closer the starting point is to the minimum, the shorter the required runtime becomes.

lsqnonlin

The optimization function *lsqnonlin* belongs to the non-linear algorithm as well and was developed for solving non-linear least-squares problems (see [1]).

SYNTAX: *x = lsqnonlin(fun,x0,x1,x2)*

fun ···	Represents the function to be minimized
x0 ···	Starting point of the optimization algorithm
x1 ···	Starting lower bound of the searching range
x2 ···	Starting upper bound of the searching range

The *lsqnonlin* function does not only provide the starting point as an input value, the lower and upper bound can also be specified by function parameters. This optimization function was tested with the same starting points as the *fminsearch* function. The obtained results are shown in table 3.8.

$x0$ [Nm/s]	Rate Limiter Value [Nm/s]	Runtime [s]
x1	82605	19.2
x2	1193911	4.5
x2/2	1193911	3.9

Table 3.8: Evaluation of the *lsqnonlin* function

It is clear immediately, that the performance of this optimization algorithm is much higher than the performance of *fminsearch*. However, it is also evident that a bad choice of the starting point increases the runtime and it might be responsible for a mismatch of the found optimum. This effect is given by the the starting point $x1 = 0$. Thereby the algorithm returns a wrong value of the function minimum. This means that the success and performance depend on the choice of the starting point.

fminbnd

The MATLAB® function *fminbnd* belongs to the non-linear optimization methods and is developed to find the minimum of a single-variable function on a fixed interval. The algorithm of this function is based on the so-called golden section search (see [2]). The golden section search is an algorithm for finding the minimum or maximum of a given function by restricting the range of values of which it is known that they contain the optimum.

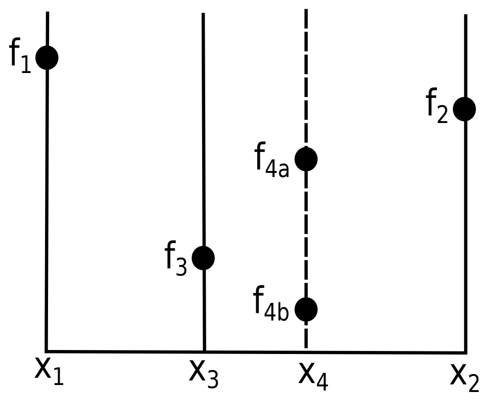


Figure 3.15: Search technique of the golden section search algorithm

The figure 3.15 illustrates the main principle in the search technique for finding a minimum. The functional values of the given function $f(x)$ are on the vertical axis and the horizontal axis represents the x parameter. The three points x_1, x_2, and x_3 of the function $f(x)$ have already been evaluated. If f_3 is smaller than f_1 and f_2, it is obvious that the minimum is inside the interval from x_1 to x_2. In the next step, a new value x_4 is evaluated within the current largest interval. In this example the value x_4 is selected in the range of x_2 and x_3. If the function value of x_4 is larger than f_3 (f_{4a}), the new range of the minimum is between x_1 and x_4. On the other hand, if the point is smaller than f_3 (f_{4b}), the searching area changes between x_2 and x_3. These steps will be repeated until the minimum of the function is found.

SYNTAX: *x = fminbnd(fun,x1,x2)*

fun ···	Represents the function to be minimized
x1 ···	Starting lower bound of the searching range
x2 ···	Starting upper bound of the searching range

This optimization algorithm does not need a starting point, only the start bounds, which automatically result from the described consideration above. Table 3.9 shows the output and performance with the start bounds *x1* and *x2*.

Rate Limiter Value [Nm/s]	Runtime [s]
1193911	4.9

Table 3.9: Evaluation of the *fminbnd* function

Table 3.9 shows that the algorithm has a good performance, like the *lsqnonlin* optimization function, and works very robustly, as the *fminsearch* algorithm. In addition to these advantages, it does not need a starting point, which may have a bad influence on the performance and result of the algorithm.
In this case, the *fminbnd* function is used for identifying the rise time parameter of the dyno

model. Equation 3.9 represents the rise time calculation and table 3.10 shows the identified rise time parameters of the engine and powertrain testbed.

$$TDynodt = \frac{Nominal\ Torque}{Rate\ Limiter\ Value} \tag{3.9}$$

	Nominal Torque [Nm]	Rate Limiter Value $[\frac{Nm}{s}]$	TDynodt [ms]
Engine Dyno	500	1193911	0.42
Powertrain Dyno 1	3000	3061224	0.98
Powertrain Dyno 2	3000	3409090	0.88

Table 3.10: Identified torque rise time of the automotive testbed dynos.

3.2 Controller Parameters

The next parameter group, which needs to be identified, are the controller parameters of the speed observer. As described in chapter 2.2.2, this parameter group consists of 6 individual parameters: The parameters *KpCurve*, *KiCurve* and *KiPhCorrCurve* are look-up tables for reducing the control parameters *Kp*, *Ki* and *KiPhCorr*. This means that if the actual dyno speed goes below a defined threshold, the controller parameters *Kp*, *Ki* and *KiPhCorr* decrease continually based on a curve progression, which can be defined by these *Curve* parameters.
This curve reduction of the control parameter is commonly applied on lower dyno speed and therefore only has an effect in driving up or down, to or from the operating speed. The gradient of the reducing curve is specified through many practical experiments and has no influence during an operating condition. In this case, only the main controller parameters *Kp*, *Ki* and *KiPhCorr* will be tuned and the characteristics of the curve parameters persist.

3.2.1 System Description

Figure 3.16 represents the implemented controlling part through a Simulink® block wiring diagram.

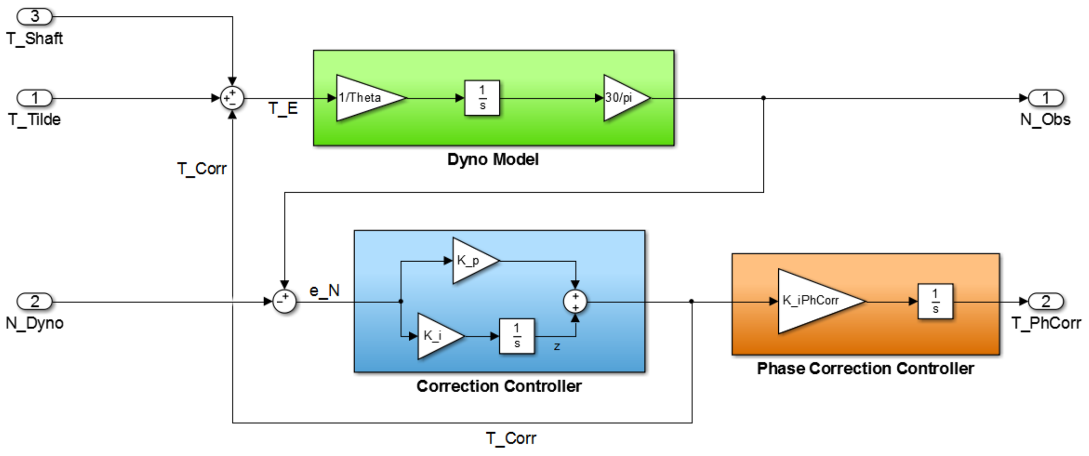


Figure 3.16: Simulink® block wiring diagram of the speed observer.

The main goal of the controlling part is to eliminate the error between measured (N_{Dyno}) and observed (N_{Obs}) speed as best and as fast as possible. The speed error is denoted by e_N and represented with the equation 3.10a. Because of the linearity of differentiation, the error equation 3.10a can be translated into the formula 3.10b.

$$e_N = N_{Obs} - N_{Dyno} \tag{3.10a}$$

$$\frac{de_N}{dt} = \frac{dN_{Obs}}{dt} - \frac{dN_{Dyno}}{dt} \tag{3.10b}$$

Furthermore, the time derivative of the error is the difference between the time derivative of the measured (N_{Dyno}) and observed (N_{Obs}) speed.
The time derivatives $\frac{dN_{Obs}}{dt}$ as well as $\frac{dN_{Dyno}}{dt}$ are described by the differential equations 3.11 and 3.12 below.

$$\frac{dn_{Obs}}{dt} = \frac{30}{\pi\ \Theta}\ T_E = \frac{30}{\pi\ \Theta}\ (\tilde{T} - T_{Corr} + T_{Shaft}) \tag{3.11}$$

$$\frac{dn_{Dyno}}{dt} = \frac{30}{\pi\ \Theta}\ (\tilde{T} + T_{Shaft}) \tag{3.12}$$

The symbol $\tilde{T}$, in the equations 3.11 and 3.12, specifies the effective torque after the delay and rise time.
Correction torque T_{Corr} represents the output value of the implemented correction controller and can be illustrated by:

$$T_{Corr} = K_p\ e_N + z \tag{3.13}$$

The symbol z of equation 3.13 demonstrates the outcome of the correction controller integration part. Therefore, its time derivative can also be illustrated as:

$$\frac{dz}{dt} = K_i\ e_N \tag{3.14}$$

The following mathematical relation of the speed error e_N results from inserting the equations 3.11, 3.12 and 3.13 into formula 3.10b:

$$\begin{aligned}\frac{de_N}{dt} &= \frac{30}{\pi\ \Theta}\left[\tilde{T} - (K_p\ e_N + z) + T_{Shaft}\right] - \frac{30}{\pi\ \Theta}\ (\tilde{T} + T_{Shaft}) \\ &= \frac{30}{\pi\ \Theta}\ (\tilde{T} - \tilde{T} - K_p\ e_N - z) \\ &= \frac{30}{\pi\ \Theta}\ (-K_p\ e_N - z)\end{aligned} \tag{3.15}$$

Based on equations 3.14 and 3.15 the following state space model with the state vector $\mathbf{x} = [e_n\ z]^T$ and the system matrix $\tilde{A}$ can be created:

$$\underbrace{\begin{bmatrix}\frac{de_N}{dt} \\ \frac{dz}{dt}\end{bmatrix}}_{\frac{d\mathbf{x}}{dt}} = \underbrace{\begin{bmatrix}-\frac{30}{\pi\ \Theta}K_p & -\frac{30}{\pi\ \Theta} \\ K_i & 0\end{bmatrix}}_{\tilde{\mathbf{A}}}\ \underbrace{\begin{bmatrix}e_N \\ z\end{bmatrix}}_{d\mathbf{x}} \tag{3.16a}$$

$$\frac{d\mathbf{x}}{dt} = \tilde{\mathbf{A}}\ \mathbf{x} \tag{3.16b}$$

On the basis of the transpose properties, the constructed system description via equation 3.16b can also be written as:

$$\left(\frac{d\mathbf{x}}{dt}\right)^T = \mathbf{x}^T\ \tilde{\mathbf{A}}^T \tag{3.17}$$

By remodelling the system matrix $\tilde{\mathbf{A}}^T$,

$$\begin{aligned}\tilde{\mathbf{A}}^T &= \begin{bmatrix}-\frac{30}{\pi\ \Theta}K_p & K_i \\ -\frac{30}{\pi\ \Theta} & 0\end{bmatrix} = \begin{bmatrix}\alpha K_p & K_i \\ \beta & 0\end{bmatrix} \\ &= \underbrace{\begin{bmatrix}0 & 0 \\ \beta & 0\end{bmatrix}}_{\mathbf{A}} - \underbrace{\begin{bmatrix}-1 \\ 0\end{bmatrix}}_{\mathbf{b}}\ \underbrace{\begin{bmatrix}\alpha K_p & K_i\end{bmatrix}}_{\mathbf{k}^T}\end{aligned} \tag{3.18}$$

the obtained state space model can also be illustrated as

$$\left(\frac{d\mathbf{x}}{dt}\right)^T = \mathbf{x}^T\ \left(\mathbf{A} - \mathbf{b}\ \mathbf{k}^T\right) \tag{3.19}$$

or

$$\frac{d\mathbf{x}}{dt} = \left(\mathbf{A} - \mathbf{b}\ \mathbf{k}^T\right)^T\ \mathbf{x} \tag{3.20}$$

This corresponds to a closed-loop system with a state feedback control of:

$$u = -\mathbf{k}^T\ \mathbf{x} \tag{3.21}$$

The correction controlling parameters K_p and K_i can be defined by calculating $\mathbf{k}^T$, which is shown with equation 3.22 below.

$$\mathbf{k}^T = \begin{bmatrix} -\frac{30}{\pi\ \Theta} K_p & K_i \end{bmatrix} \tag{3.22}$$

There are many different techniques for calculating the feedback vector of a state controller. A common, widespread tuning technique is the so-called pole placement. Location of closed-loop poles correspond directly to the eigenvalues of the system, which specify the characteristics of the system response. One disadvantage of this tuning technique is, that with the choice of the poles it is hard to keep an overview of the control and state values behaviour (see [9] for further details).
A design concept, which offers a good influence on the control and state behaviour, is the linear-quadratic regulator (abbr. LQR). This method calculates the control parameters by minimizing a quadratic target function and it belongs to the optimal control concepts.

3.2.2 Linear-Quadratic Regulator

This chapter describes the design concept of a linear-quadratic regulator. Primarily, at the beginning of this chapter, some important control-specific definitions are described for a better understanding of the LQR problem and its solution conditions.

3.2.2.1 Definite Matrix

According to [7], a symmetric $n \times n$ matrix $\mathbf{A}$ with only real entries is

- positive definite, if $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$
- positive semidefinite, if $\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$
- negative definite, if $\mathbf{x}^T\mathbf{A}\mathbf{x} < 0$
- negative semidefinite, if $\mathbf{x}^T\mathbf{A}\mathbf{x} \leq 0$

for all real non-zero vectors $\mathbf{x}$.
These conditions can be satisfied with an eigenvalue inspection of the matrix $\mathbf{A}$. Positive values of all eigenvalues of the given matrix mean that it is positive definite. If one or more eigenvalues are zero, the matrix will be a positive semidefinite matrix. The same applies for negative definite/semidefinite matrices, if the inspection of the eigenvalues is carried out with inverse sign matrix.
If the matrix is a diagonal matrix, the entries of the main diagonal illustrates its eigenvalues. Therefore, the definite of the matrix can easily be identified by the diagonal values.

3.2.2.2 Controllability and Observability

Controllability and observability are important properties of a control system for developing a desired control concept.
A given system is controllable, if any system input exists to move the internal states $\mathbf{x}$ of the system from any initial state to any other final state in finite time T [9]. At this definition of controllability, an exact description of the system input is not considered. It is merely important that any input $\mathbf{u}$ is available, which fulfils the condition in finite time ($\mathbf{x}(0) \rightarrow \mathbf{x}(T)$).

The controllability of a system can be proved by the so-called *Kalman* criterion (see [9]) . The controllability matrix $\mathbf{S}_u$ has to be regular.

$$\mathbf{S}_u = \begin{pmatrix} \mathbf{b} & \mathbf{A}\mathbf{b} & \mathbf{A}^2\mathbf{b} & \dots & \mathbf{A}^{n-1}\mathbf{b} \end{pmatrix} \tag{3.23}$$

$$det(\mathbf{S}_u) \neq 0 \tag{3.24}$$

The observability of a system is given, if the unknown initial state $\mathbf{x}(0)$ can be specified with the input $u(t)$ and output $y(t)$ behaviour in a finite time interval [0,T]. This system property can also be proved by a criterion of *Kalman* (see [9]). In this case, the so-called observability matrix B_y must be regular.

$$\mathbf{B}_y = \begin{pmatrix} \mathbf{c}^T \\ \mathbf{c}^T\mathbf{A} \\ \mathbf{c}^T\mathbf{A}^2 \\ \vdots \\ \mathbf{c}^T\mathbf{A}^{n-1} \end{pmatrix} \tag{3.25}$$

$$det(\mathbf{B}_y) \neq 0 \tag{3.26}$$

3.2.2.3 Solution of the LQ-Problem

As described in [13], the general problem definition is to compute a state controller

$$\mathbf{u}(t) = -\mathbf{k}^T\mathbf{x}(t) \tag{3.27}$$

where $\mathbf{u}$ represents the control and $\mathbf{x}$ the state of the given LTI system

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x}(t) + \mathbf{b}\mathbf{u}(t), \qquad \mathbf{x}(0) = \mathbf{x}_0 \tag{3.28}$$

The determined controller stabilizes the closed-loop system and minimizes the objective function J:

$$J = \int_0^\infty (\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{u}^T\mathbf{R}\mathbf{u})\,\mathrm{d}t \overset{!}{=} \min \tag{3.29}$$

The first step for solving the optimization problem is to set up the equation of the objective function for the closed-loop system. The function J is defined as

$$J = \int_0^\infty (\mathbf{x}(t)^T(\mathbf{Q} + \mathbf{k}^T\mathbf{R}\mathbf{k})\mathbf{x}(t))\,\mathrm{d}t \tag{3.30}$$

and the closed-loop system is represented by the following equation:

$$\frac{d\mathbf{x}}{dt} = (\mathbf{A} + \mathbf{b}\mathbf{k}^T)\mathbf{x}, \qquad \mathbf{x}(0) = \mathbf{x}_0 \tag{3.31}$$

By a given $\mathbf{k}^T$ and known initial states, $\mathbf{x}(t)$ can be calculated by:

$$\mathbf{x}(t) = e^{(\mathbf{A}+\mathbf{b}\mathbf{k}^T)t}\mathbf{x}_0 \tag{3.32}$$

By inserting formula 3.32 into the equation 3.30, the following relation results:

$$\begin{aligned} J &= \int_0^\infty \mathbf{x}_0^T e^{(\mathbf{A}+\mathbf{b}\mathbf{k}^T)t}(\mathbf{Q} + \mathbf{k}\mathbf{R}\mathbf{k}^T)e^{(\mathbf{A}+\mathbf{b}\mathbf{k}^T)t}\mathbf{x}_0\,\mathrm{d}t = \\ &= \mathbf{x}_0^T \underbrace{\left(\int_0^\infty e^{(\mathbf{A}+\mathbf{b}\mathbf{k}^T)t}(\mathbf{Q} + \mathbf{k}\mathbf{R}\mathbf{k}^T)e^{(\mathbf{A}+\mathbf{b}\mathbf{k}^T)t}\,\mathrm{d}t\right)\mathbf{x}_0}_{\mathbf{P}} = \\ &= \mathbf{x}_0^T\mathbf{P}\mathbf{x}_0 \end{aligned} \tag{3.33}$$

The expression $\mathbf{P}$ of formula 3.33 is the solution of the following Lyapunov equation:

$$(\mathbf{A} + \mathbf{b}\mathbf{k}^T)^T\mathbf{P} + \mathbf{P}(\mathbf{A} + \mathbf{b}\mathbf{k}^T) + \mathbf{Q} + \mathbf{k}\mathbf{R}\mathbf{k}^T = 0 \tag{3.34}$$

Generally, Lyapunov equation appears in many different subject areas of control theory, like stability analysis or in optimal control. The theorem states that for the Lyapunov equation

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} + \mathbf{Q} = 0 \tag{3.35}$$

exists a unique $\mathbf{P} \succ 0$ by any $\mathbf{Q} \succ 0$ exists, if the system $\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x}$ is globally asymptotically stable. When these conditions are given the unique solution of $\mathbf{P}$ is represented by this equation:

$$\mathbf{P} = \int_0^\infty e^{\mathbf{A}^T t}\mathbf{Q}e^{\mathbf{A}t}\,\mathrm{d}t \tag{3.36}$$

It quickly becomes clear that the equation 3.33 has the same structure as the solution of the Lyapunov equation (see formula 3.36). Thus, it can be concluded that this equation 3.33 is the solution of a Lyapunov equation with the structure of equation 3.34. Furthermore, the equation 3.34 can be rewritten in the following form:

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} - \mathbf{P}\mathbf{b}\mathbf{R}^{-1}\mathbf{b}^{\mathbf{T}}\mathbf{P} + \mathbf{Q} + (\mathbf{P}\mathbf{b}\mathbf{R}^{-1} + \mathbf{k})\mathbf{R}(\mathbf{R}^{-1}\mathbf{b}^T\mathbf{P} + \mathbf{k}^T) = 0 \tag{3.37}$$

The feedback control $\mathbf{k}^T$ is only confined to the term:

$$(\mathbf{PbR}^{-1} + \mathbf{k})\mathbf{R}(\mathbf{R}^{-1}\mathbf{b}^T\mathbf{P} + \mathbf{k}^T) = 0 \tag{3.38}$$

In this case $\mathbf{k}^T$ results from the expression:

$$\mathbf{k}^T = -\mathbf{R}^{-1}\mathbf{b}^T\mathbf{P} \tag{3.39}$$

This reduces the equation 3.37 to:

$$\mathbf{A}^T\mathbf{P} + \mathbf{PA} - \mathbf{PbR}^{-1}\mathbf{b}^T\mathbf{P} + \mathbf{Q} = 0 \tag{3.40}$$

In the literature, the obtained equation 3.40 is referred to as *Riccati* equation. For this reason, the linear-quadratic regulator is often called the *Riccati* regulator. The *Riccati* equation produces the optimal control matrix $\mathbf{k}^T$,

$$\mathbf{u} = -\mathbf{k}^T\mathbf{x} \tag{3.41a}$$

$$\mathbf{k}^T = \mathbf{R}^{-1}\mathbf{b}^T\mathbf{P} \tag{3.41b}$$

if the following control conditions are satisfying (see [13] theorem 7.2, page 298):

- Matrix $\mathbf{Q}$ has to be symmetric and positive semidefinite ($\mathbf{Q} \succeq 0$)
- Matrix $\mathbf{R}$ has to be symmetric and positive definite ($\mathbf{R} \succ 0$)
- The system $[\mathbf{A}, \mathbf{b}]$ must be completely controllable
- The system $[\mathbf{A}, \mathbf{Q_1}]$ must be completely observable ($\mathbf{Q} = \mathbf{Q}_1^T\mathbf{Q}_1$)

The first two conditions ($\mathbf{Q} \succeq 0$ & $\mathbf{R} \succ 0$) refers to the choice of the matrices $\mathbf{Q}$ and $\mathbf{R}$. It must be ensured that the definite conditions of chapter 3.2.2.1 are satisfied.
To check the controllability ($[\mathbf{A}, \mathbf{b}]$) of the given system, the criterion of *Kalman* for the dyno observer system must be performed as follows:

$$\mathbf{S}_u = \begin{bmatrix} -1 & 0 \\ 0 & -\alpha \end{bmatrix} \tag{3.42a}$$

$$det(\mathbf{S}_u) = (-1)\ (-\alpha) = \alpha \tag{3.42b}$$

$$\alpha \neq 0 \Rightarrow -\frac{30}{\pi\ \Theta} \neq 0 \tag{3.42c}$$

From the equations above it can be easily derived, that the given system is controllable, because the term α can never be zero. In this case the LQR condition, which $[\mathbf{A}, \mathbf{b}]$ must be controllable, is satisfied.
The last condition for producing the optimal control matrix $\mathbf{k}^T$, by solving the *Riccati* equation, is to check the observability of $[\mathbf{A}, \mathbf{Q_1}]$, whereas the matrix $\mathbf{Q_1}$ results from the segmentation:

$$\mathbf{Q} = \mathbf{Q}_1^T\mathbf{Q}_1 \tag{3.43}$$

If the matrix $\mathbf{Q}$ is positive definite ($\mathbf{Q} \succ 0$) instead of positive semidefinite ($\mathbf{Q} \succeq 0$), the last condition will always be true (see [13]). In this case, the matrix $\mathbf{Q}$ will always be set to a positive definite matrix.

3.2.3 Choosing LQR Weights

As described in [13], the essential part of a LQR-designed regulator is the choice of the matrix $\mathbf{R}$ and $\mathbf{Q}$ for solving the target function 3.29. $\mathbf{R}$ and $\mathbf{Q}$ are symmetric matrices, which define the proportion between control and state values. The matrix $\mathbf{R}$ controls the energy expenditure of the received optimal controller.

- $\mathbf{R} \ \rightarrow\ 0$: expensive control, i.e high controller manipulation value
- $\mathbf{R} \ \rightarrow\ \infty$: cheap control, i.e low controller manipulation value

Changes of the matrix $\mathbf{Q}$ have an influence on the state values, whereas each state behaviour is individually adjustable by the different diagonal values of the matrix. The other elements do not influence the state behaviour. Because of the same characteristic, the matrix $\mathbf{R}$ is suitable for the manipulation value of the controller. This is an additional reason why the matrices $\mathbf{Q}$ and $\mathbf{R}$ are usually diagonal matrices. A increase of the diagonal values $\mathbf{Q}$ in relation to $\mathbf{R}$ results a faster transient oscillation of the system state by higher manipulation values of the controller. In contrast, a increasing of $\mathbf{R}$ in relation to $\mathbf{Q}$ leads to a slower but more energy-efficient system. In principle, it is a iterative process to find the proper values of the matrices $\mathbf{R}$ and $\mathbf{Q}$. First, the matrices entries are set to any individual values. Afterwards, the controller should be calculated by solving the *Riccati* equation, and then the behaviour of the closed-loop system needs to be analysed. This procedure must be repeated until the desired system behaviour is generated.
The described matrices definition procedure was also performed for calculating the control parameters of the speed observer. If the existing system is a SISO system (single-input single-output system), like the given system here, the matrix $\mathbf{R}$ will be a scalar. In this case, to comply with the condition, the scalar R has to be a real value which is grater than zero ($R > 0 | R \in \mathbb{R}$). From now, the described matrix $\mathbf{R}$ is referred to as scalar R, because of the considered speed observer SISO system.
For the definition procedure of the matrix $\mathbf{Q}$ and the scalar R, the recorded quantities of the speed and torque jump test run (see chapter 3.1.1 and 3.1.2) from the engine testbed were used as the input values of the dyno observer. The behaviour of the observer was analysed by constant diagonal values of matrix $\mathbf{Q}$ and individual value of R, because the resulting system behaviour depends on the relation between theses two parameters. Additionally, this approach simplifies the analysis by tuning only one parameter.
Figures 3.17 and 3.18 illustrate the comparison between real measured dyno speed and observer speed by different R values and the corresponding correction controller output. The appropriate matrix $\mathbf{Q}$ was a diagonal matrix with the value 100 for both diagonal entries ($\mathbf{Q} =$ diag([100,100])). It can be realized that the output of the correction controller increase when the value of R is increased as well. In return, the observed speed characteristic agrees better with the measured speed by an “expensive“ control produced by a low R value. It can also be seen that by “low“ control ($R = 1$), the observed speed characteristic also agrees well with the measured dyno speed. The reason for this is that the identified model parameters are a good fit for modelling the real dyno behaviour. Therefore, the correction controller does not need a significant correction for adapting the observed speed. It follows that by decreasing the parameter R, the torque correction values grow enormously, but the effective improvement is not essential.

Thus, it does not make sense to further decrease the value of R, because there is no additional benefit. Furthermore, the correction controller will reinforce unwanted system dynamics on high control parameters at an extreme rate. That might cause unstable control behaviour.
For these reasons, the parameter R is not set extremely low for an "expensive" control.

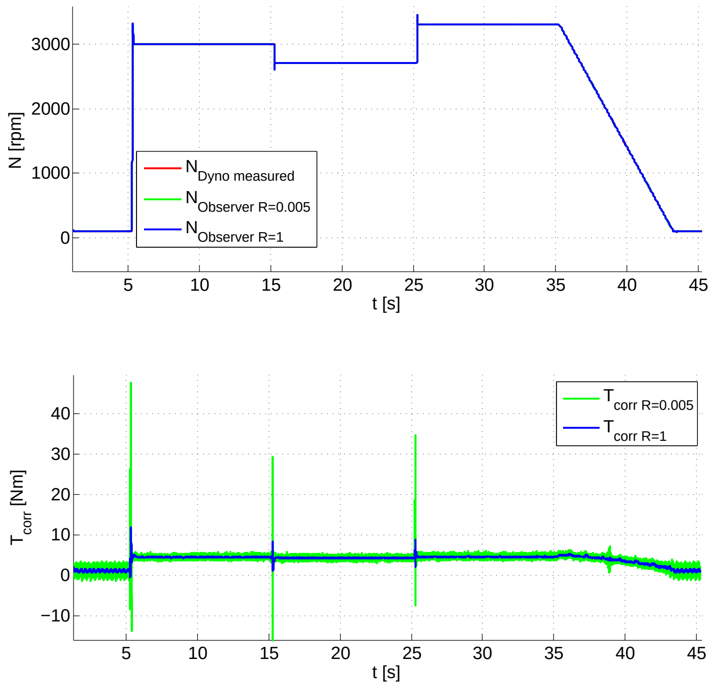


Figure 3.17: Comparing individual values of R using the speed test run.

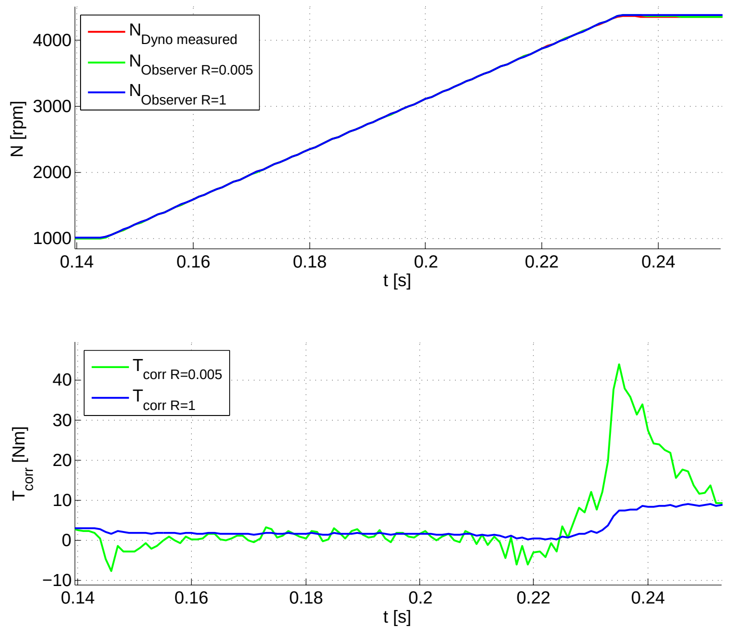


Figure 3.18: Comparing individual values of R using the torque jump test run.

By default, the matrix $\mathbf{Q}$ is set to a diagonal matrix with the value 100 for both diagonal entries

($\mathbf{Q}$ = diag([100,100])) and the scalar R is set to one, like in the figures 3.17 and 3.18 above. The feedback vector $\mathbf{k}^T$ can be calculated with the help of the MATLAB® function *lqr*. After that, the controlling parameter K_p and K_i can be calculated by rewriting the $\mathbf{k}^T$ vector. On the basis of the engine testbed, the calculation reads as follows:

$$\begin{aligned}
\mathbf{k}^T &= lqr(\mathbf{A}, \mathbf{b}, \mathbf{Q}, R) \\
&= lqr(\begin{bmatrix} 0 & 0 \\ -11.9366 & 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}, 0.005) \\
&= \begin{bmatrix} -152.8927 & 141.4214 \end{bmatrix} \\
\mathbf{k}^T &= \begin{bmatrix} -\frac{30}{\pi\,\Theta} K_p & K_i \end{bmatrix} \\
\Longrightarrow K_p &= 12.8087 \\
K_i &= 141.4214
\end{aligned}$$

The matrix $\mathbf{Q}$ and the scalar R are set to the values described above by default and cannot be changed by the testbed engineer for the identification process. However, the proper identification software, which is specified in chapter 5, offers a feature for changing the calculated K_p and K_i parameters with the help of a validation plot after the identification procedure. As already mentioned, the parameter for the phase correction controller will always be recommended as one-tenth of the calculated K_i value from the correction controller.

3.3 Filter Parameters

The filters are used to filter out undesired dynamic effects, which should not be coupled back into the control unit. These undesired effects are usually the resonance frequencies of the given mechanical system which occur during operation. According to [3], resonance occurs, if a system oscillates with an increasing amplitude from a periodic stimulation with the natural frequency of the given system. Every mechanical system has certain natural frequencies that depend on the physical conditions of the system. In this case, it is essential to analyse the testbed in the common operation condition. For the identification of the filter parameters, the unit under test must be coupled to the corresponding dynos.
To identify the natural frequencies of the automotive testbed construction, it is necessary to stimulate the system with the full frequency range. A so-called chirp signal is ideal for this request. This signal changes its frequency and stimulates the system with the needed frequency range.
Afterwards, a frequency research of this stimulation via a *fast Fourier transformation* needs to be performed to detect the natural frequencies of the system. Then, the filter parameters can be defined with the help of this analysis.
The following chapters 3.3.1 and 3.3.2 deal with the theoretical foundations of the chirp signal and the discrete Fourier transformation via *FFT*. Furthermore, they provide some information on the technical implementation of these concepts.
Chapter 3.3.3 treats the analyses of the chirp test runs which are executed on automotive testbeds for calculating the speed and torque filter parameters. Additionally, it describes the advantages of this identification method over the previous approach for parameters identification.

3.3.1 Chirp Signal

A signal which changes its frequency over time is referred to as chirp signal (see [11]). The name "chirp" is a reference to the chirping sound of birds. This kind of signal is usually used in sonar and radar systems for locating objects or rather detecting their velocity.
The general mathematical description of a chirp signal is represented by equation 3.44. In formula 3.44, $f(\tau)$ stands for the time dependent frequency at which the primitive of $f(\tau)$ needs to be inserted in the integral. Symbol Φ_0 illustrates the initial phase at time $t = 0$ and symbol A represents the amplitude of the chirp signal.

$$x(t) = A\ sin\left(2\pi \int_0^t f(\tau)d\tau + \Phi_0\right) \tag{3.44}$$

A frequently used type of chirp signal is the *linear chirp.* Instantaneous frequency of $f(\tau)$ modifies linearly in progress. Equation 3.45a describes this linear modification, where as f_0 is the start frequency and k represents the chirp rate. The calculation of the linear chirp rate is shown by formula 3.45b. Here, f_1 represents the final frequency and T stands for the time interval between the start frequency f_0 and the end frequency f_1. Finally, the linear time response *x(t)* (see equation 3.45c) results by inserting equation 3.45a into the general chirp description formula 3.44.

$$f(\tau) = f_0 + k\tau \tag{3.45a}$$

$$k = \frac{f_1 - f_0}{T} \tag{3.45b}$$

$$\begin{aligned} x(t) &= A\ sin\left(2\pi \int_0^t f(\tau)d\tau + \Phi_0\right) \\ &= A\ sin\left(2\pi \int_0^t (f_0 + k\tau)d\tau + \Phi_0\right) \\ &= A\ sin\left(2\pi\ (f_0 t + \frac{k}{2}t^2) + \Phi_0\right) \end{aligned} \tag{3.45c}$$

Chirp signals with exponential variation of the frequency are commonly used for sonar and radar systems. The variation function is shown by equation 3.46a. Like before, f_0 stands for the start frequency and k illustrates the chirp rate (see equation 3.45b). The resulting exponential time response *x(t)* is represented below, by formula 3.46b.

$$f(\tau) = f_0 k^\tau \tag{3.46a}$$

$$\begin{aligned} x(t) &= A\ sin\left(2\pi \int_0^t f(\tau)d\tau + \Phi_0\right) \\ &= A\ sin\left(2\pi\ f_0 \int_0^t k^\tau d\tau + \Phi_0\right) \\ &= A\ sin\left(2\pi\ \frac{f_0(k^t - 1)}{ln(k) + \Phi_0}\right) \end{aligned} \tag{3.46b}$$

The figures below represent a comparison of the described linear and exponential chirp signals. Figure 3.19 shows the linear, and figure 3.20 the exponential chirp. Both chirp signals are executed with the same parameters. The chirps start with the frequency $f_0 = 1Hz$ and finish with $f_1 = 20Hz$. The time interval T is 1 second.

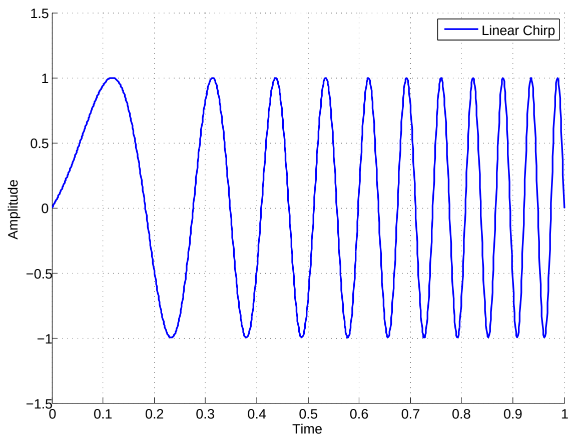


Figure 3.19: Linear chirp signal.

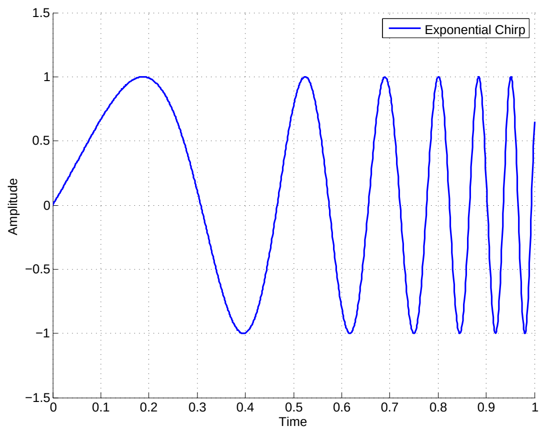


Figure 3.20: Exponential chirp signal

3.3.2 Fast Fourier Transformation

The Fourier theorem, developed by Jean Baptiste Joseph Fourier, states that any periodic function can be expressed as the sum of series with sine or cosine terms, which are called the Fourier series (see [5]). This theorem is the basis for the Fourier analysis which includes different variants of signal transformation. Generally, Fourier transformation deconstructs a time domain representation of a signal into the frequency domain representation.
Equation 3.47, taken from [14], chapter 3.1.1, shows the common calculation of a discrete Fourier transformation (abbr. DFT). This kind of Fourier transformation converts a discrete time signal x_n from its time domain into the frequency domain. This transformation is used, because the recorded testbed quantities are only available as discrete time sequence values.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi nk}{N}} \qquad k = 0, 1, 2, \ldots, N-1 \qquad (3.47)$$

Closer observation of this formula 3.47 shows, that the DFT computation of N data samples needs N^2 complex multiplications and $N(N-1)$ complex summations. In this case, the time and data storage required for computing increase as a quadratic function depending on the number of recorded values. This computing problem of the DFT was solved by James Cooley and John Turkey. In 1965, they released their calculation algorithm of a DFT named *fast Fourier transformation* (FFT). This algorithm exploits the symmetries of the discrete Fourier transformation. The calculation of the value X_{N+k} looks as follows:

$$\begin{aligned} X_{N+k} &= \sum_{n=0}^{N-1} x_n \; e^{-\frac{j2\pi n(N+k)}{N}} \\ &= \sum_{n=0}^{N-1} x_n \; \underbrace{e^{-j2\pi n}}_{1} \; e^{-\frac{j2\pi nk}{N}} \\ &= \sum_{n=0}^{N-1} x_n \; e^{-\frac{j2\pi nk}{N}} \end{aligned} \tag{3.48}$$

As shown by equation 3.47 and 3.48, the calculations of X_k and X_{N+k} are the same. Therefore, the following can be concluded:

$$X_{\alpha N+k} = X_k \tag{3.49}$$

This symmetry of the DFT calculation provides the basis for the FFT algorithm of James Cooley and John Turkey (see [14], chapter 6). They divide the DFT computation into two smaller subpartitions:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \; e^{-\frac{j2\pi n(N+k)}{N}} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} \; e^{-\frac{j2\pi k(2m)}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} \quad e^{-\frac{j2\pi k(2m+1)}{N}} \\ &= \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m} \; e^{-\frac{j2\pi km}{N/2}}}_{\text{even-numbered values}} + e^{-\frac{j2\pi k}{N}} \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} \; e^{-\frac{j2\pi km}{N/2}}}_{\text{odd-numbered values}} \\ &= E_k + e^{-\frac{j2\pi k}{N}} O_k \end{aligned} \tag{3.50}$$

That separation in odd- and even-numbered values does not reduce any computational cycles, but the knowledge of the existing DFT symmetry (see equation 3.49) shows that only half of the computations for each subproblem need to be performed.

$$E_k = E_{k+\frac{N}{2}} \tag{3.51a}$$

$$O_k = O_{k+\frac{N}{2}} \tag{3.51b}$$

Each subproblem can be divided in further subproblems of even- and odd-numbered values and so on. This divide-and-conquer approach can be implemented as an efficient recursive computational algorithm. The FFT algorithm of James Cooley and John Turkey is also implemented in MATLAB® and it is used for the frequency analyses of the chirp test run on the automotive testbed.
The result of the FFT has a complex format with real and imaginary parts. Therefore, it is

possible to visualise the magnitude and the associated phase of each frequency component. The magnitude of the frequency component is an important piece of information for the determination of the filter parameters. Therefore, considerations of individual signals taken in this section focus on the magnitude of the FFT.

The fast Fourier transformation is an efficient algorithm for calculating a discrete Fourier transformation. It converts a finite periodic discrete signal from its time or space domain into a frequency domain. In this manner, the FFT is used for a number of different applications, such as compression of digital images, mobile technologies and noise detection. Figure 3.21 represents a simple sinusoidal signal and the corresponding FFT. The frequency domain shows one peak at 10 Hz with a magnitude of 1. This illustrates that the analysed signal consists only of one component with a frequency of 10 Hz and an amplitude of 1. Thus, it is a perfect sinusoidal signal without additional frequencies. In this example, further peaks at other frequencies would indicate a noise-containing signal. Therefore, the FFT provides a comfortable way for analysing the frequency spectrum of a discrete signal and for the development of filters, which are used for the elimination of noise or other undesirable frequency components.

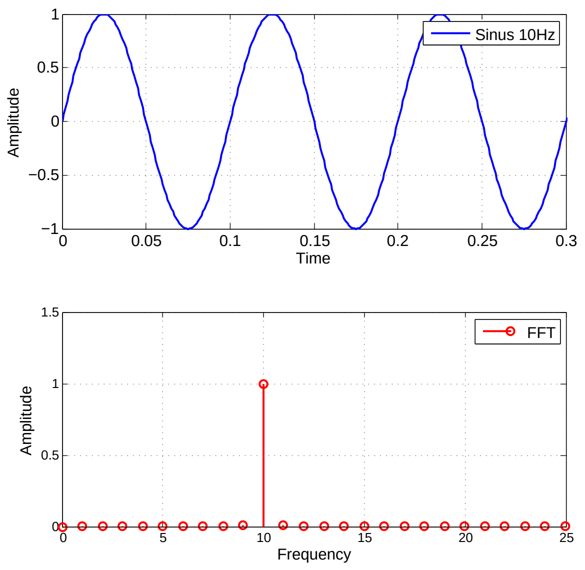


Figure 3.21: Ideal FFT representation of a sinus wave.

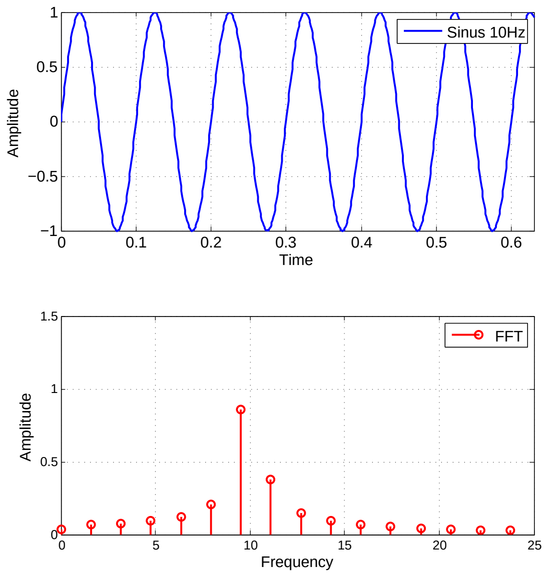


Figure 3.22: Spectral leakage due to noninteger number of periods.

Due to the limitations of the FFT, using its algorithm for causally recorded signals can yield wrong results. The FFT algorithm assumes that the existing finite discrete signal points describe one period of a periodic signal. This is like the start and end point of the time domain signal are connected for creating an endless circle signal topology. Therefore, it is a non-negligible problem, if the signal contains discontinuity at the start or end point. The discontinuity will be leading to additional frequency components in the frequency domain. These supplemental frequencies are much higher than the Nyquist frequency and they are aliased in the range of zero to the half sampling rate. This unwanted effect, caused by a finite viewing window, is known as spectral leakage. The consequence of a spectral leakage is illustrated in figure 3.22. An FFT of the same signal as before was performed with the only difference, that the observed section does not end in a cycle period. It can be recognized immediately that additional frequencies are detected by the FFT, the magnitude of the main frequency coasts out into other frequencies

and the whole signal magnitude increases as well.
With the so-called windowing technique, spectral leakage of an FFT over aperiodic signals can be reduced. The idea of windowing is to reduce the amplitude of the discontinuous parts of the signal. By windowing, the time domain signal is multiplied by a window containing a varying amplitude which goes towards zero at the boundaries. The multiplication results in a signal with a continuous form at the endpoints. Therefore, the discontinuous parts do not have that much influence on the transformation.

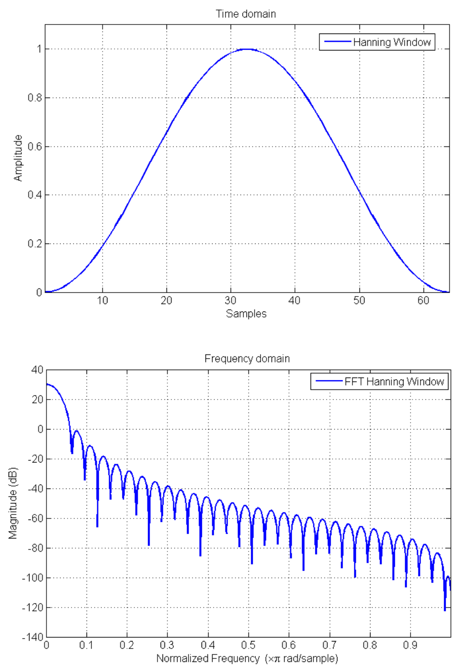


Figure 3.23: Hanning window.

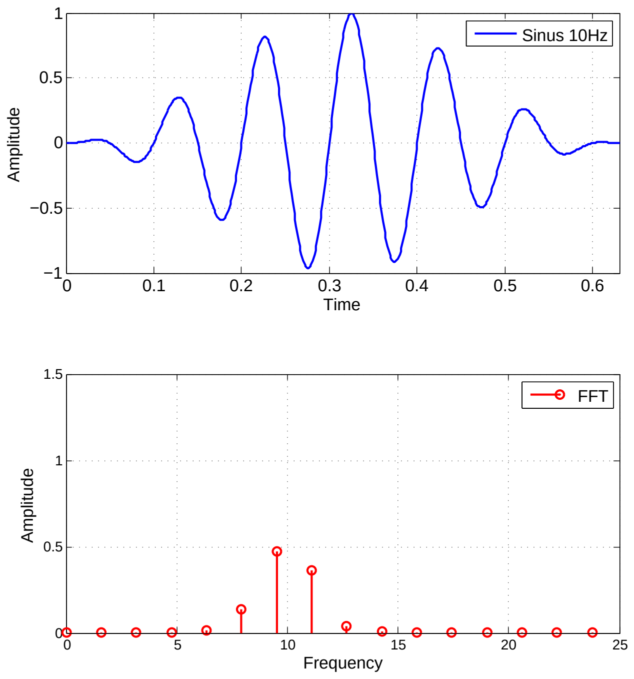


Figure 3.24: FFT with Hanning window.

There are different window functions. The type of window function to be used depends on the current signal. Generally, a window function consists of a centred main lobe and side lobes around it, which approach zero. Side lobes characteristics define the elimination of the spectral leakage and manipulate the bandwidth of the main lobe. Lower side lobes imply a reduction of the spectral leakage, but increase the bandwidth and vice versa. Therefore, the choice of the window function has a significant impact on the result. Selecting the ideal window function for a given signal requires detailed analysing, which requires a lot of time.
A corresponding elaborate analysis cannot be executed during the identification process. For the filter parameter identification an approved window function according to [10] is deployed. This function is shown in figure 3.23: The so-called Hanning window. The Hanning window main lobe is centred by the time signal and the side lobes go down to zero. It reduces spectral leakage excellently. This window function is well-known for good results while the nature of the signal is unknown, so it is also ideal for analysing the chirp test speed and shaft torque signal for calculating the corresponding filter parameters.
The figure 3.24 represents the Hanning windowed FFT of the same aperiodic signal as before in figure 3.22. It is clear that the effect of the spectral leakage is reduced significantly, the correct main frequency becomes an advantage and the magnitude coasts out only around the main frequency.
However, this windowing technique reduces spectral leakage but it also provides individual amplitude weighting of the different signal parts as a side effect. This side effect is not a significant

problem for a simple signal, like in figure 3.24, but it must be considered for more complex signals. Especially, when analysing variable signals to find out the most vigorous resonance frequency, it is important to have the same weighting for every frequency component to achieve a meaningful and informative comparison. To correct this undesired side effect, the signal to be examined needs to be split in smaller overlapping subviewing parts for which the windowing function is used. The overlapping of the windowing function reduces the time-depending weighting of the signal that is produced by the windowing. The Hanning window needs an overlapping of exactly 50 % to eliminate the amplitude reduction. An overlapping of several Hanning windows and their resulting window is represented below in figure 3.25.

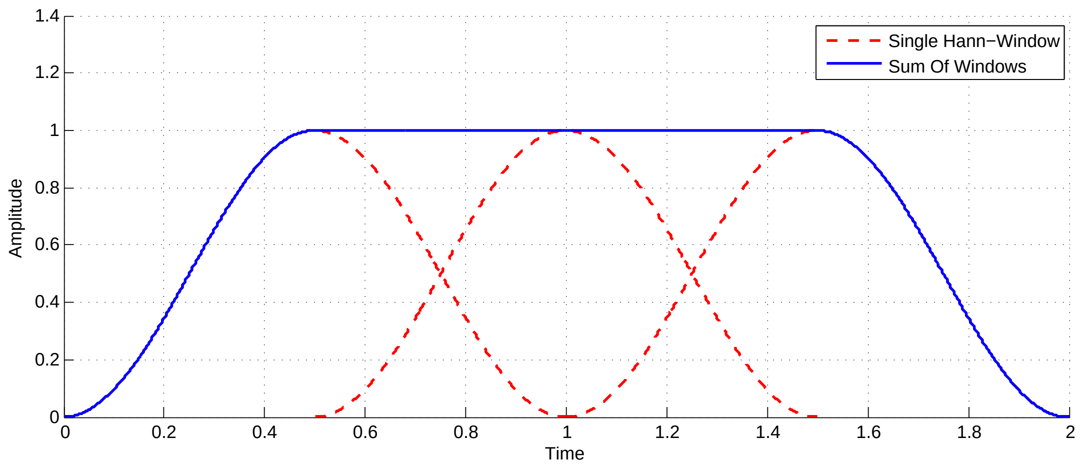


Figure 3.25: Hanning window overlapping of 50%

Another important aspect when operating a DFT is the time density of the given signal and the frequency density of the corresponding transformation. The continuous time-based signal is recorded by N samples, which are also used for the frequency domain representation. A sample frequency of f_s results in a time density of Δt between each recording point:

$$\Delta t = \frac{1}{f_s} \tag{3.52}$$

The sampling rate is specified by the control system. Therefore, the time density Δt cannot be changed. The DFT of the recorded samples can be calculated with 3.47. In this equation, x_i $(0 \leq i \leq N-1)$ represents the N samples. The result X_k $(0 \leq k \leq N-1)$ stands for the frequency domain representation of x_i. Similarly to the time density Δt, there is a corresponding frequency density Δf in the frequency domain representation. The frequency density Δf describes the frequency separation between the components of X and can be calculated by:

$$\Delta f = \frac{f_s}{N} = \frac{1}{N\Delta t} \tag{3.53}$$

Equation 3.53 illustrates, that the frequency density can be modified by varying the sampling rate f_s or by varying the number of samples N. The time and frequency density are linked by the sampling rate f_s and are influenced by each other. Especially for audio analysis and technology, a compromise between time and frequency resolution needs to be found. As already mentioned before, the sampling rate is defined by the control system. Therefore, time density

Δt is not changeable. This is why the the frequency density can only be changed through the number of record values N. In this case, the size of the Hanning window specifies the frequency density. Splitting the recorded signal in smaller overlapping viewing windows, like in figure 3.25, provides not only a correct amplitude over the main area, but it also allows for the frequency density to be adjusted.
These received signal parts are averaged to one frequency spectrum by the so-called *Averaging*. The implemented *Averaging* process for analysing the chirp test run, is based on the described mechanism of [1] in chapter 6.2.2. During*Averaging*, the signal is fragmented into individual parts with the length of the corresponding window size. After that, the FFT is performed for every individual signal fragment and the results are summed up to one sequence with the length of the viewing window. This *Averaging* technique is used to reduce spectral leakage and for averaging the signal noise over the viewing window to reduce its undesirable influence. The only custom adaptations of that described algorithm in [1] are that the individual signal parts are windowed by the Hanning window for an additional reducing of the spectral leakage and the signal fragments are overlapped by 50 % for correcting the amplitude.

3.3.3 Filter Parameter Identification

The testbed controlling software *EMCON* contains a configurable signal generator which provides different signal waveforms. The generator also includes a chirp signal, which can be adapted by eight parameters for changing the signal characteristics. These parameters are described in table 3.11. The implemented chirp signal is shown in figure 3.26. Its frequency increases exponentially in the first half and decreases with the same rate in the second half. It is therefore an exponential chirp signal, which passes through the frequency range in an up and down chirp.

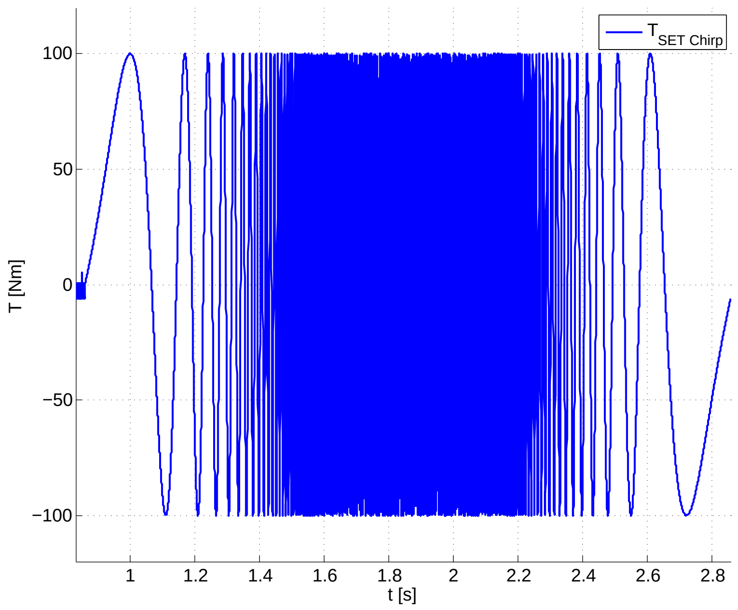


Figure 3.26: Implemented chirp signal for filter parameters identification.

As opposed to the speed and torque jump test run, the unit under test is connected to the dynos during the chirp test run. The speed and torque limitations of the coupled engine or powertrain need to be considered.

There are many different units under test with individual configurations. Therefore, a general statement of the exact test run parameters cannot be made. The commissioning engineer is invited to bring the testbed to a commonly used and stable state, in which the testbed will be further operated. An important aspect is that the dynos must be controlled to a constant speed value.
If the testbed runs in this stable state, the chirp signal will be executed. It has a length of approximately two seconds and an amplitude of about 10 % of the maximum dyno torque with an offset of 50 %. The output mode (see table 3.11) is set to 1, which defines that the signal generator overwrites the output of the dyno controller. Otherwise (output mode = 0) the generator output would be added to the controller output and, therefore, the controller would operate against this chirp signal to keep the dynos at a constant speed. As a result of that control signal overwriting, the speed value would fall down rapidly at the beginning of chirp signal, caused by the unit under test and reversed torque values. The chirp signal amplitude offset is necessary to counter this speed drop a bit. It is important that the speed does not go down to zero to get an unusual effect instead of the resonance frequencies of the testbed.
If the further chirp test run analysis provides a non-satisfying result, the test run should be repeated with a higher amplitude for a stronger system stimulation.
Equation 3.54 describes the output of the function generator for the chirp signal mathematically. The function generator parameters *StartFrequency*, *MaxFrequency* and *Duration* are needed to calculate the chirp rate k by the formula 3.45b. Parameter *StartFrequency* represents f_0, parameter *MaxFrequency* illustrates f_1 and formula symbol T of equation 3.45b is defined by the function generator parameter *Duration.* Sign A of equation 3.54 symbolizes the *Amplitude* parameter of the function generator, T_{Offset} represents the parameter *Offset* and Φ_0 is the initial phase shift, which is illustrated via parameter *InitialPhaseShift.* The function generator parameter *NumberOfExecutions* is a loop parameter for repeating the chirp signal output.

Parameter	**Description**
`OutputMode`	Defines the output mode of the signal generator. 0 ... Generator output is added to the controller output 1 ... Generator output replaces the controller output.
`StartFrequency`	Defines the start frequency of the chirp signal.
`MaxFrequency`	Sets the end frequency of the chirp signal.
`Amplitude`	Determines the amplitude of the chirp signal.
`Offset`	Describes the amplitude offset from zero.
`Duration`	Defines the length of the chirp signal parameter.
`NumberOfExecutions`	Defines the numbers of chirp test executions.
`InitialPhaseShift`	Defines the initial phase shift of the outgoing chirp signal.

Table 3.11: Chirp signal parameters of the signal generator.

$$T_{FcnGen} = (A + T_{Offset})\ sin\left(2\pi\ \frac{f_0(k^t - 1)}{ln(k)} + \Phi_0\right) \tag{3.54}$$

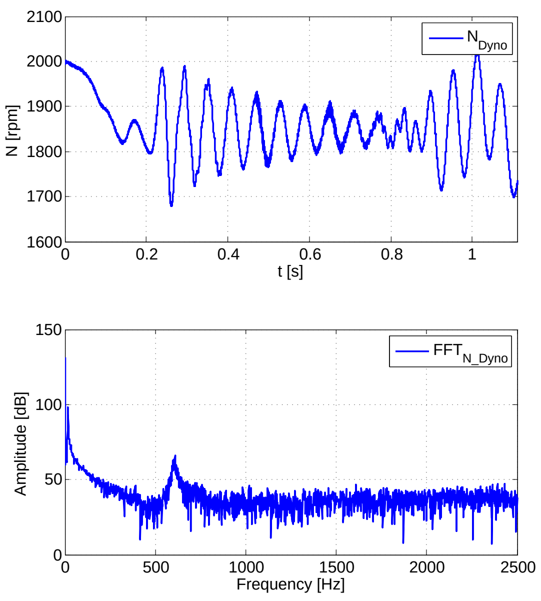


Figure 3.27: FFT of the speed signal.

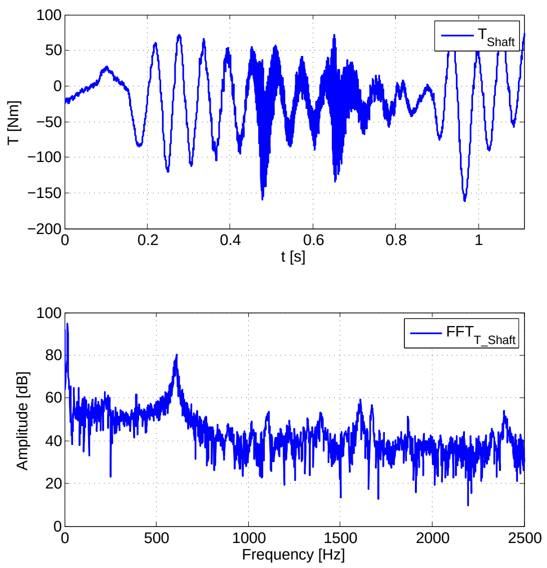


Figure 3.28: FFT of the shaft torque signal.

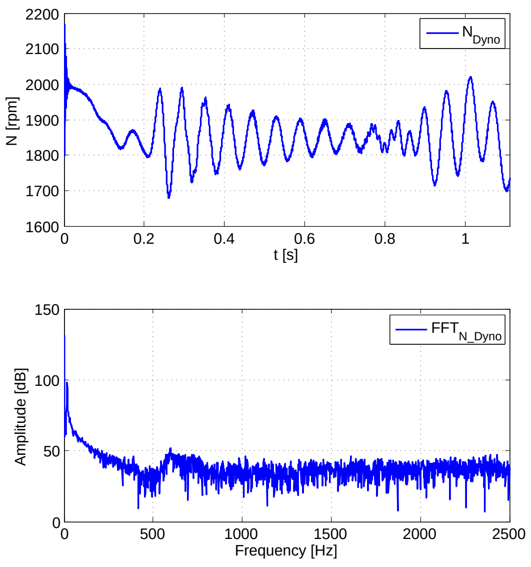


Figure 3.29: FFT of the fildered speed signal.

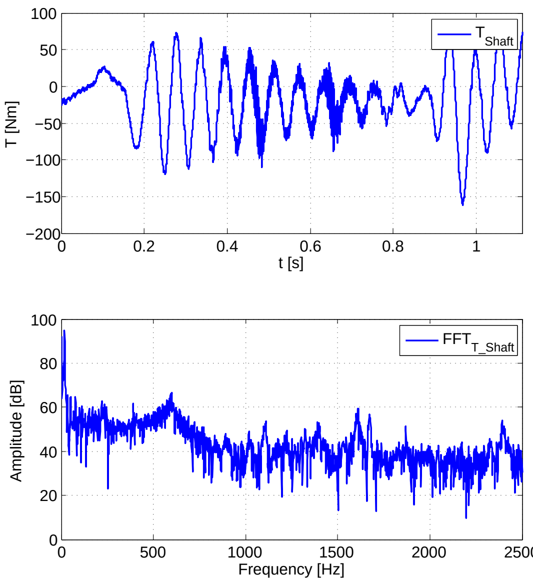


Figure 3.30: FFT of the filtered shaft torque signal.

Figures 3.27 and 3.28 illustrate the recorded speed and shaft torque signals of an engine testbed by executing the chirp test run with the corresponding FFT plot. As described previously, the speed drops at the beginning of the chirp signal and shows strong variation in spite of a chirp torque offset. For the test run, it is important to choose a proper speed value. Furthermore, both signals show a resonance frequency of about 610 Hz.
Generally, the observer filters are implemented as a second order digital *IIR* filter. These filters

can be operated either as low-pass filters or as notch filters depending on the corresponding filter configuration. The low-pass filter can be used for minor dynamic requirements. In this case, frequencies that exceed the determined resonance frequency are also filtered out. The cut-off frequency of the low-pass filter should be slightly below the resonance frequency in order to filter it out it effectively. For requirements that are more dynamic, like the requirements of the speed observer, the notch filter should be used. The trap frequency of the notch filter must be set to the exact resonance frequency. A notch filter with the highest resonance frequency is always selected for the parameterisation of the observer filters. If different resonance frequencies are dominant at the measured shaft torque and speed, each filter can be set differently.
In this case, both filters are notch filters with a trap frequency of 610 Hz. The recorded signals filtered by the notch filters are shown above. It is visible that the filters have damped the resonance frequency successfully and the desired effect occurs.
A Hanning window with the size of $N = 4096$ was used for the FFT in the figures 3.27 and 3.28 above. Because the sampling rate was 5 kHz, the following frequency density is given:

$$\Delta f = \frac{1}{N\Delta t} = \frac{1}{4096 \cdot 0.0002} \approx 1.22Hz \tag{3.55}$$

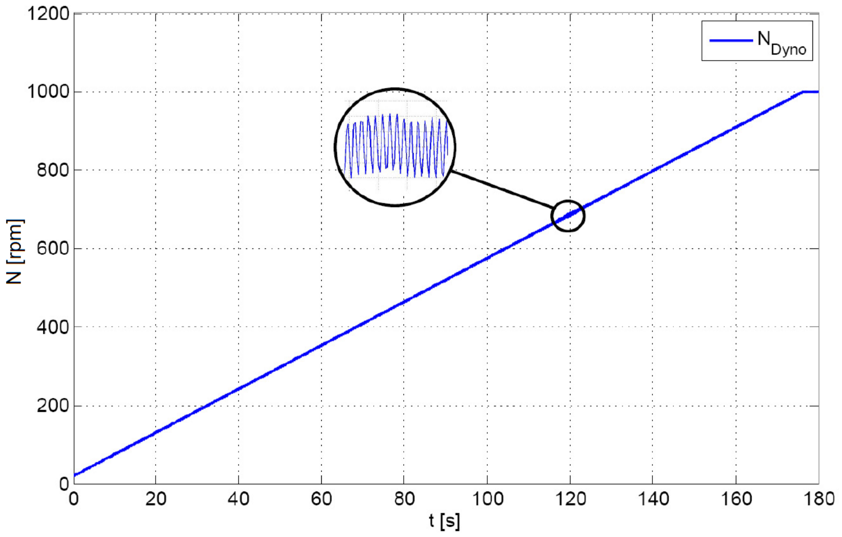


Figure 3.31: Slow speed ramp for resonance frequency detection.

Currently, testbed engineers use a slow speed ramp for determining the filter parameters. If a resonance frequency is stimulated by the control signal during the ramp test run, the speed signal shows a higher variation. This phenomenon is illustrated in figure 3.31. The frequency of the resonance is reflected by the higher speed variation and conforms with that variation frequency. Therefore, the engineers set the notch filter trap frequency to the identifying speed variation frequency at this point.
The occurrence of more than one speed variation during the speed ramp indicates the existence of more natural frequencies of the mechanical system. This is a great problem for the mentioned identification method. Since there is only one filter for filtering out undesired effects, it cannot be decided objectively, which resonance frequency has a greater interference effect.
The uncertainty that all frequencies have been excited and stimulated with the same strength evokes another identification problem. Therefore, the individual natural frequencies cannot be compared effectively and it is not sure, whether each frequency is visible during the speed ramp.

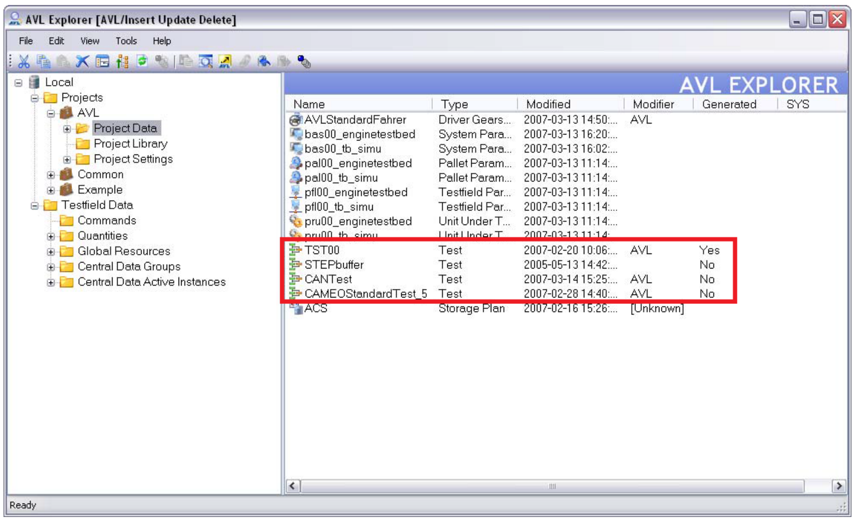
Compared to this identification technique, the chirp test run with an FFT provides several advantages. Each ascertainable frequency is excited with approximately the same strength. Thus, all existing resonance frequencies are visible in the speed and shaft torque signal. With the help of the FFT, the resonance frequencies can be detected easily and their strengths are comparable. Furthermore, the chirp signal represents a short and compact identification test run which can be performed within a few seconds, whereas the identification on basis of the speed ramp takes several minutes.

4 PUMA Test Run Environment

The automation software *PUMA Open* already provides a comfortable editor for the generation of automotive testbed test runs. This chapter gives a short introduction of this editor and describes the created identification test runs in detail.

4.1 Test Run Editor

The *PUMA Open* software suite includes many interaction and management tools for creating and manipulating testbed projects. One of these tools is the *AVL Explorer*, which is shown below in figure 4.1. This application combs through the *PUMA Open* workspace and displays all AVL-specific files in a clear and ordered structure. The files which specify automatic test runs are among these files and they are highlighted with the red rectangle in figure 4.1. The test run editor can be started by double clicking on one of the predefined test run files. The *AVL Explorer* offers an export and import feature for all AVL file types. It is therefore possible to create a custom test run which can be shared and used at other automotive testbeds.



Name	Type	Modified	Modifier	Generated	SYS
AVLStandardFahrer	Driver Gears...	2007-03-13 14:50:...	AVL		
bas00_enginetestbed	System Para...	2007-03-13 16:20:...			
bas00_tb_simu	System Para...	2007-03-13 16:02:...			
pal00_enginetestbed	Pallet Param...	2007-03-13 11:14:...			
pal00_tb_simu	Pallet Param...	2007-03-13 11:14:...			
pfl00_enginetestbed	Testfield Par...	2007-03-13 11:14:...			
pfl00_tb_simu	Testfield Par...	2007-03-13 11:14:...			
pru00_enginetestbed	Unit Under T...	2007-03-13 11:14:...			
pru00_tb_simu	Unit Under T...	2007-03-13 11:14:...			
TST00	Test	2007-02-20 10:06:...	AVL	Yes	
STEPbuffer	Test	2005-05-13 14:42:...		No	
CANTest	Test	2007-03-14 15:25:...	AVL	No	
CAMEOStandardTest_5	Test	2007-02-28 14:40:...	AVL	No	
ACS	Storage Plan	2007-02-16 15:26:...	[Unknown]		

Figure 4.1: AVL Explorer.

If the test run editor is started, a window, as shown in figure 4.2, opens. This test run window contains two subwindows. The subwindow on the left side of the test run editor is the *Toolbox*. The *Toolbox* contains predefined components, which can be used in a custom test run. These components are combined into groups and can be added to the right subwindow per drag and drop. The right subwindow is called *BSQ Editor*, whereas “BSQ“ is an acronym for **B**lock **S**e**Q**uence. The *BSQ Editor* represents the window, where the testbed test run can be built and configured.

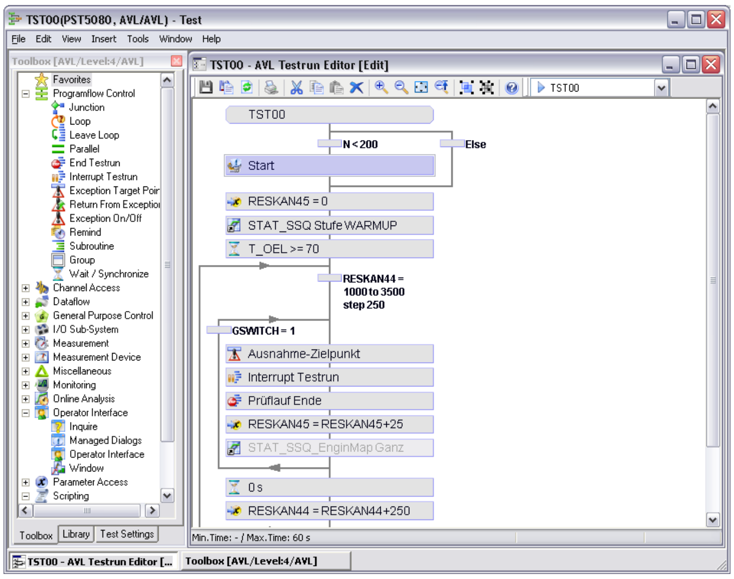


Figure 4.2: User interface of the test run editor.

The *Toolbox* contains several predefined components, but only the ones which are used in the identification test run will be discussed in more detail in the following section.

4.1.1 Toolbox Tab

For a better overview, the components of the *Toolbox* are structured into several tabs. The *Toolbox* tab *Program Flow* contains blocks, which control the flow of the test run. The following table describes the used *Program Flow* components of the identification test run.

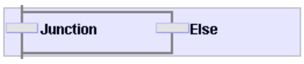
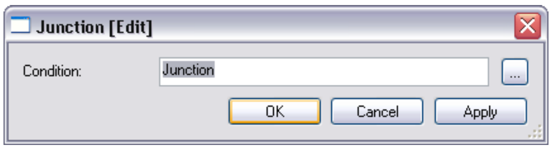
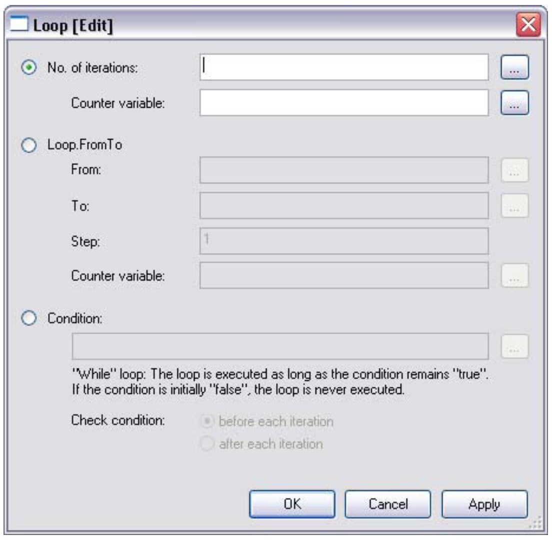
Test run block	Description
`Junction` 	Depending on the junction condition, the test run branches to the left path (condition is true) or right path (condition is false). Via right click, more junctions (conditions) can be defined. The condition can be changed by double-clicking the junction symbol.
`Loop`	Components in a loop can be executed multiple times, either by running a fixed number of iterations or a number of iterations that depend on a condition. For selecting a loop variation, an editing window (see figure 4.4) will open by double-clicking the loop symbol.
`End Test Run`	If the test run reaches this block, it will be interrupted and the automotive testbed persists in the current state.

Table 4.1: Used *Program Flow* test run blocks.


Junction [Edit]

Condition: Junction

Figure 4.3: Editing window for junction condition.


Loop [Edit]

[✓ No. of iterations:]
Counter variable:

Loop.FromTo
From:
To:
Step: 1
Counter variable:

Condition:

"While" loop: The loop is executed as long as the condition remains "true".
If the condition is initially "false", the loop is never executed.

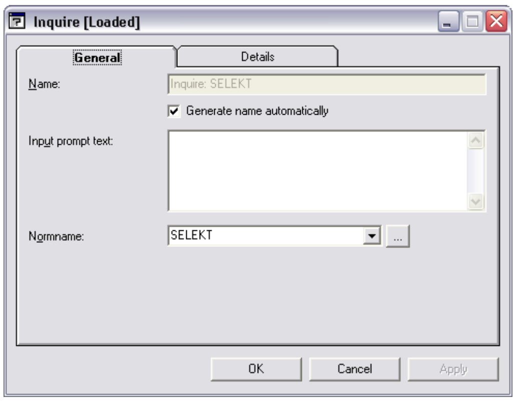
Check condition: [✓ before each iteration] after each iteration

Figure 4.4: Editing window for loop block.

The *Toolbox* tab *Operator Interface* contains some test run blocks for interaction with the testbed operator. These components allow for manipulation of testbed settings during the automatic test run and offer to control the test run. The table below shows the used *Operator Interface* components.

Test run block	**Description**
`Inquire`	This component opens a simple standardised dialogue window with a prompt text (see figure 4.5) and a single input box. Further, it is possible to define a time out and a range for the input value, which is shown in figure 4.6.
`Managed Dialogs`	This predened component opens a defined *PUMA* dialogue window during the test run. The *PUMA* automation software provides a design function mode in which the user can design custom *PUMA* windows and dialogues. Thus, every input or information window can be built and opened during a test run for interaction.

Table 4.2: Used *Operator Interface* components.


Inquire [Loaded]

General | Details

Name: Inquire: SELEKT

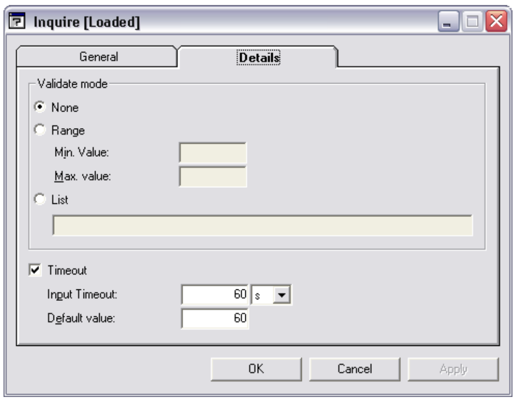
☑ Generate name automatically

Input prompt text:

Normname: SELEKT

OK Cancel Apply

Figure 4.5: Inquire General settings window.


Inquire [Loaded]

General | Details

Validate mode

● None

○ Range

Min. Value:

Max. value:

○ List

☑ Timeout

Input Timeout: 60 s

Default value: 60

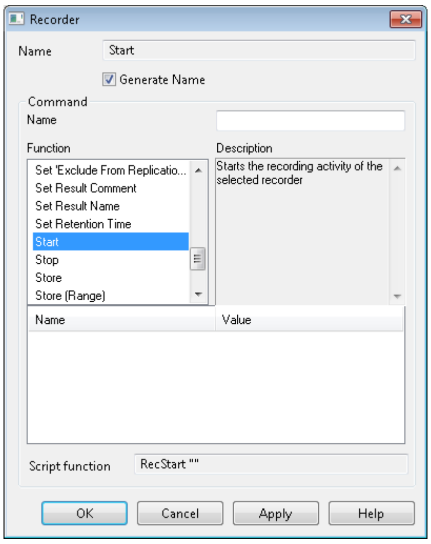
OK Cancel Apply

Figure 4.6: Inquire Details settings window.

An essential task is the measurement and record function of the UUT and testbed values. In this case, the *Toolbox* provides many measure and record components which are grouped in the *Measurement* tab. The test run blocks for controlling the UUT and testbed dynos are also important components. These blocks can be found under the *UUT Control* tab.

Test run block	**Description**
`Recorder`	This test run component enables to record all available quantities of the testbed. The record object and proper function command can be defined via the record settings window, which is illustrated in figure 4.7.

Table 4.3: Used *Measurement* components.


Recorder

Name: Start

☑ Generate Name

Command

Name

Function: Set 'Exclude From Replicatio... / Set Result Comment / Set Result Name / Set Retention Time / Start / Stop / Store / Store (Range)

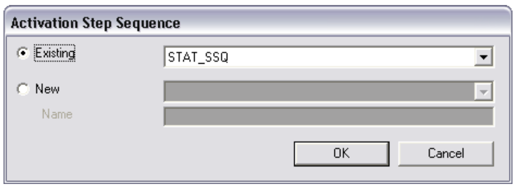
Description: Starts the recording activity of the selected recorder

Name | Value

Script function: RecStart ""

OK Cancel Apply Help

Figure 4.7: Recorder settings window.


Activation Step Sequence

● Existing STAT_SSQ

○ New

Name

OK Cancel

Figure 4.8: Step sequence settings window.

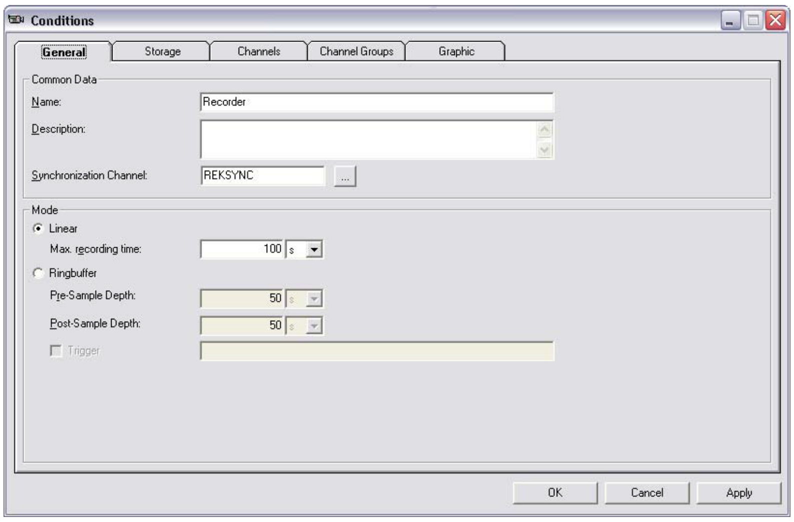
Test run block	Description
`Step Sequence`	A *Step Sequence* defines a testbed activity, which consists of one or more individual steps. The performed activity depends on the *Step Sequence* object.

Table 4.4: Used *Measurement* components.

The record and the step sequence *Toolbox* component (see table 4.3 and 4.4) needs a block object, which must be defined in the *Library* tab of the *Toolbox* subwindow. These objects will be described in the following chapter 4.1.2.

4.1.2 Library Tab

Some test run components need a runtime object for executing. Such objects can be defined in the *Library* tab of the *Toolbox* (see figure 4.2). Right-clicking in the Library tab opens the context menu for creating a new component object.
A *Record* object defines the necessary information for the tracing function of the *Record* test run component. The quantities, which need to be recorded, the recording mode and a lot of other recording settings can be defined in the settings window (see figure 4.9) of the *Toolbox* object. For using the recording object, the defined object name must be inserted in the *Name* field of the *Record* test run component (see figure 4.7).



Common Data

Name: Recorder

Description:

Synchronization Channel: REKSYNC

Mode

(●) Linear

Max. recording time: 100 s

( ) Ringbuffer

Pre-Sample Depth: 50 s

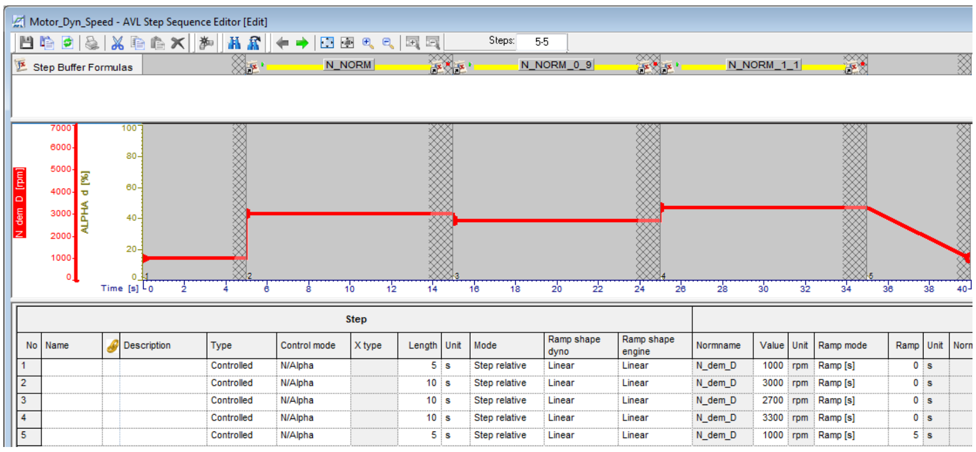
Post-Sample Depth: 50 s

☐ Trigger

Figure 4.9: Recording object settings.

As already mentioned, the *Step Sequence* block also needs a defined library object. There are several different *Step Sequence* objects, which can be chosen at context menu. Depending on the testbed type and desired request the corresponding *Step Sequence* objects have to be chosen. Figure 4.10 represents the settings dialogue of a *Step Sequence* object for an engine testbed.
Generally, the settings window is divided into four sections from top to bottom. First, the function toolbar is located at the top of the settings window. The component beneath the

toolbar is called *Action Control*, which is used to create action tracks. In each of the action tracks, specific step actions or sequence actions can be defined. Step actions are actions, which are valid for an individual step only. Sequence actions are defined for an entire range of steps. Therefore, with this *Action Control*, it is possible to execute some individual activities during a step.
Under the *Action Control* there is a graphical visualization of the different step actions. This figure example represents an engine testbed *Step Sequence* object, so in this case the course of the dyno speed and the alpha position of the engine throttle are shown in this visualization part.
The fourth and last section is called *Grid Control*. The *Grid Control* displays and contains all steps and their data in table form, clearly arranged in groups. It provides functions for convenient input of demand values, for example the demand speed value of the dyno or the alpha position of the engine throttle. The *Grid Control* part also provides the possibility to add or remove sequence steps.

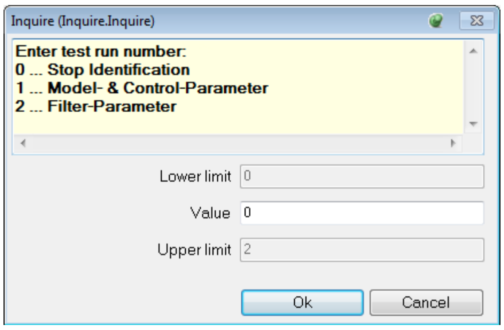


Step																			
No	Name		Description	Type	Control mode	X type	Length	Unit	Mode	Ramp shape dyno	Ramp shape engine	Normname	Value	Unit	Ramp mode	Ramp	Unit	Norm
1				Controlled	N/Alpha		5	s	Step relative	Linear	Linear	N_dem_D	1000	rpm	Ramp [s]	0	s	
2				Controlled	N/Alpha		10	s	Step relative	Linear	Linear	N_dem_D	3000	rpm	Ramp [s]	0	s	
3				Controlled	N/Alpha		10	s	Step relative	Linear	Linear	N_dem_D	2700	rpm	Ramp [s]	0	s	
4				Controlled	N/Alpha		10	s	Step relative	Linear	Linear	N_dem_D	3300	rpm	Ramp [s]	0	s	
5				Controlled	N/Alpha		5	s	Step relative	Linear	Linear	N_dem_D	1000	rpm	Ramp [s]	5	s	

Figure 4.10: Step sequence object.

4.2 Identification Test Run

The first step of the observer parameter identification process is to import the predefined identification test run in the *AVL Explorer*. A test run is only executable with *PUMA Open* if it is available at the *AVL Explorer*. After the import procedure, the *PUMA Open* software should be started up for operating the identification test run.
The test run starts with an inquiry about which observer parameters should be identified. The user has the option to stop the identification, to execute speed and torque jump test run for the model and control parameter identification or the chirp test run to identify the filter parameter (see figure 5.11). By choosing to identify the model and control parameter group, an extra warning dialogue will open. The window, as shown in figure 4.12, points out that all dynos have to be decoupled from the unit under test, otherwise it will get damaged. This measure ensures that the testbed user is definitely informed about the consequences.
After confirming via *OK* button, the test run scaling parameters need to be entered in a graphical user interface window, which is shown in figure 4.13. The values of the different parameters are suggested by the appropriate identification software. The handling and interaction with this identification software is described in the following chapter 5. As soon as the test run parameter mask is filled out and confirmed, the speed and torque jump test run will be performed.

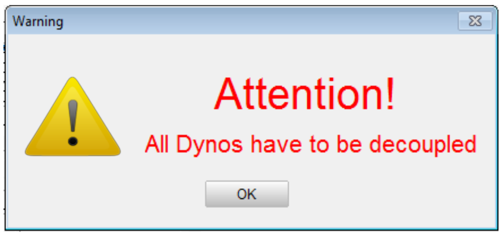


Inquire (Inquire.Inquire)

Enter test run number:
0 ... Stop Identification
1 ... Model- & Control-Parameter
2 ... Filter-Parameter

Lower limit	0
Value	0
Upper limit	2

Figure 4.11: Test run selection window.

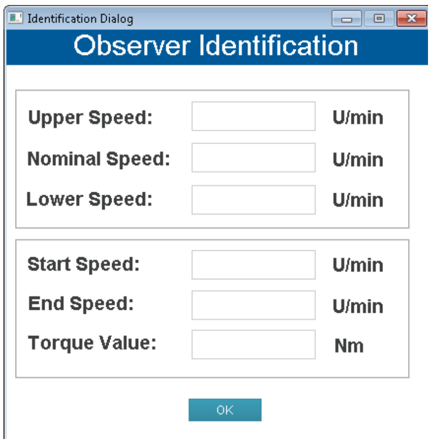


Warning

Attention!
All Dynos have to be decoupled

Figure 4.12: Dyno decouple warning.

After the execution, one record file for the speed and one for the torque jump test run will be produced. These record files can be analysed by the identification software for calculating the individual observer parameter. Finally, the test run repeats and the inquiry window will be shown again.
If the user enters number 2 for identifying the filter parameter, the dialogue mask, which is shown in figure 4.14, will appear afterwards. The user is responsible for ensuring that the testbed is in a common stationary state before the filter parameters identification test run is selected. This must be done before, because the user is not able to control the testbed manually during an automatic test run.
As before, the test run parameter for the chirp test run will be specified by the identification software. Then, the chirp test will be performed and a record file for the identification software will also be created for detecting the filter parameters.
As previously described, the test run will start again and the user can stop the identification test run by entering the default number zero.

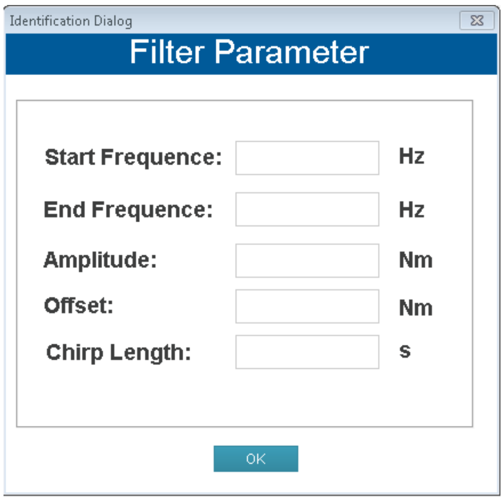


Identification Dialog

Observer Identification

Upper Speed:		U/min
Nominal Speed:		U/min
Lower Speed:		U/min
Start Speed:		U/min
End Speed:		U/min
Torque Value:		Nm

Figure 4.13: Dialogue window for speed and torque jump test run.



Identification Dialog

Filter Parameter

Start Frequence:		Hz
End Frequence:		Hz
Amplitude:		Nm
Offset:		Nm
Chirp Length:		s

Figure 4.14: Dialogue window for chirp test run.

5 Identification Software

The analysis mechanism of the test runs for the observer parameters identification is implemented via MATLAB® scripts and Simulink® models. On this account, the whole identification software, logic and GUI, is created via MATLAB®. The developed *Matlab* files are compiled to a standalone executable by the MATLAB® *Compiler* package, which can be used on the *PUMA* computer in a simple way.
This chapter deals with the description of the identification scripts implementation and with the build and usage of the identification software named *ObTune*.

5.1 Identification Scripts

An identification MATLAB® script was implemented for each observer parameter. These scripts do not have any dependencies among each other, which leads to the fact that every script can be used alone to calculate the corresponding observer parameter. This has the advantage that the whole identification process does not stop if a problem occurs while calculating one of the observer parameters.
The calculation of the inertia and the optimization of the control parameters are the only exceptions. Before the inertia can be calculated, the speed-dependent operating loss is needed (see chapter 3.1.2). The operating loss value will be assumed to be zero, if this parameter is missing during the inertia identification. Therefore, the dyno inertia is needed for the determination of the controller parameters. If this parameter is missing because of some identification error, the inertia will be defined as 1 kgm^2 for the optimization. If such exceptions happen, the MATLAB® scripts will return a corresponding error message for indicating the identification adoptions.
The identification observer parameters and other important results are stored in a *.mat*-file. This file acts as the data storage of the identification software and contains all necessary information for the software handling.

5.1.1 Operational Losses Function

The function header of the dyno losses identification MATLAB® function looks as follows:

SYNTAX: *[Operational_Loss,T_F_ARRAY, Error, Error_Str] = getOperationalLosses(T_SET,N_SET,N_DYNO,N_ARRAY,TIME_VECTOR,i)*

The function parameter *T_SET,N_SET* and *N_DYNO* represent the recorded quantities of the speed test run (see chapter 3.1.1). The quantity *T_SET* corresponds to the output of the speed controller, N_SET to the demand speed value and N_DYNO to the measured dyno speed. The proper time sequence of these recorded signals has to be handed over by the *TIME_VECTOR*

parameter. The *N_ARRAY* parameter is an array with the three demand speed values for calculating the operation losses, and parameter i represents the number of the dyno for which the identification is carried out. This is necessary for mapping the plots to the identified parameters for the visualisation by the graphical user interface.
This identification function offers four return parameters: The *Operational_Loss* parameter specifies the identified operation loss and the *T_F_ARRAY* contains the individual operating loss of every demand speed. The *Error* parameter signalises a defect of the identification mechanism. The reason for the identification error will be transmitted by the *Error_Str* return value of the *getOperationalLosses* MATLAB$^{®}$ function.
First of all, the identification script searches the start and end point of the first speed value from the *T_F_ARRAY* in the demand speed signal. After that, the starting point is postponed half of the determined time section. Then, the value of the measured speed signal at this new start point is checked in order to find out if it is in a defined range around the demand speed. If it is not in this range, the start point will be postponed until the speed value is in the range.
If a valid starting point is found, the *T_SET* values will be averaged in this time section, which specifies the operation loss by this speed value. This process is repeated for the other demand speed values. After that, these three calculated operation loss values of the individual demand speeds are averaged to one operational loss value, which represents the *Operational_Loss* parameter. If an error occurs during this mechanism, the script stops with a proper error message.
The complete identification information will be saved in a data structure and shown in a MATLAB$^{®}$ plot (see figure 3.2) of the identification software.

5.1.2 Dyno Inertia Function

The following function is used for calculating the dyno inertia:

SYNTAX: *[THETA,SLOPE, Error, Error_Str] =*
getInertia(N_DYNO,T_SET,T_JUMP,N_ARRAY,FRICTION,TIME_VECTOR,i)

The function parameters *N_DYNO* and *T_SET* represent the recorded quantities of the torque jump test run (see chapter 3.1.2). The function parameter *T_JUMP* defines the torque step value, *T_F_ARRAY* already contains the individual demand speeds, the parameter *FRICTION* specifies the preassigned operational loss and the *TIME_VECTOR* represents the time sequence of the testbed test run. As in the *getOperationalLosses* function, the function parameter i is used for mapping the plots to the identified parameters in the data structure.
The *getInertia* MATLAB$^{®}$ function returns to the identified dyno inertia with the parameter *THETA*, if no error occurs during the calculation method. Otherwise, the *Error* return parameter is set to a non-zero value, the inertia *THETA* is defined as zero and the *Error_Str* parameter contains an information of the error reason. The *SLOPE* return parameter represents the slope of the speed signal during the torque step in the range of $\pm 20\%$ around the nominal speed. This parameter is also set to zero, if an error occurs. It is only used for a plausibility check and debugging purposes.
The first action of the *getInertia* function is to find the points where the speed value is higher than 80 % and 120 % of the nominal speed for the first time. The slope of the measured speed signal is calculated between this point via linear regression (see chapter 3.1.2). After this has been successfully performed, the dyno inertia can be calculated by the equation 3.5.
This inertia detection will be illustrated in figure 3.4 for the user in the identification software.

5.1.3 Delay Time Function

The system time delay can be determined if the following function call is executed:

SYNTAX: *[DELAY_TIME,DELTA_N,Error, Error_Str] =*
getDelayTime(N_DYNO,T_SET,TIME_VECTOR,i)

As in the other identification function, the *N_DYNO* parameter represents the measured speed signal and the function parameter *T_SET* defines the dyno input torque value of the recorded test run. As already mentioned, this identification uses the quantities of the torque jump test run. The *TIME_VECTOR* parameter provides the time information and i illustrates the dyno number for which the identification is performed.
The return function parameter *DELAY_TIME* contains the detected reaction time of the electrical machine, the *Error* parameter signalises identification problems and *Error_Str* shows the corresponding error message. The return value *DELTA_N* displays the increasing speed between the torque jump start point and the first noticed speed change. This additional parameter is used for debugging purposes.
At first, the function identifies the start point of the torque step. After that, the mechanism calculates the average variance of the measured speed from the demand value of the constant speed part before the torque jump point. This variance, increased by a certain percentage, indicates the threshold for the output change detection. If this speed threshold is passed, the function defines this measurement point as the end of the delay time. The time interval between these points represents the delay time and the speed difference is the *DELTA_N* parameter of the *getDelayTime* function.
The whole identification information is saved in a data structure and it is displayed in the identification software in a MATLAB® plot, which is shown in figure 5.1. The delay time measurement points (start and end point) are illustrated with magenta coloured dots for a visual feedback.

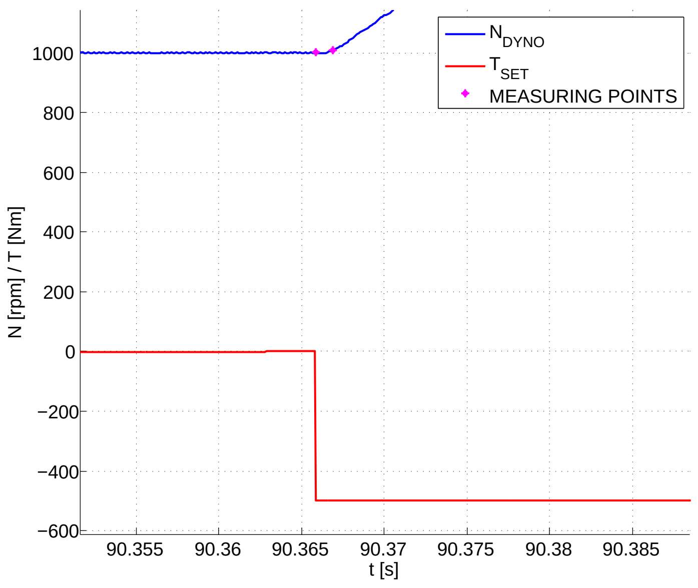


Figure 5.1: Illustration of the calculated delay time.

5.1.4 Torque Rise Time Function

The function for the identification of the torque rise time has a different structure than the other identification functions, because this parameter is calculated via an optimization algorithm (see chapter 3.1.4). The function header of this identification MATLAB® function looks as follows:

SYNTAX: *[TORQUE_RATE,Error, Error_Str] = getTorqueRate(i)*

The *getTorqueRate* uses only one function parameter *i*, which represents the dyno identifier for the mapping in the data structure. In this function the optimization algorithm *fminbnd(@ratefunction,x1,x2)* is called with the additional MATLAB® function *ratefunction*. *Ratefunction* returns the error between the measured and simulated speed signal. The simulated speed signal is generated by the Simulink® dyno model. The parameters of the dyno model are taken from the data structure, which contains the identified model parameters as well as the recorded test run quantities. The torque signal from the torque jump test run is used as the input for the dyno model and the identified parameters are deployed for the Simulink® model parameters. The torque rise time parameter is first specified on the lower limit by the optimization function.
After the simulation, the speed error is calculated and the rise time parameter will be changed as long as the error becomes a minimum. The optimization algorithm stops and the *getTorqueRate* returns the calculated rise time by the *TORQUE_RATE* return parameter. If an unexpected error occurs, the *Error* parameter will become a non-zero value, the error reason will be transmitted by the *Error_Str* parameter and the *TORQUE_RATE* will be set to zero.

5.1.5 Control Parameter Function

As already discussed in chapter 3.2, the observer parameters of the correction controller are specified through the *LQR* method. This controller design method is already implemented in MATLAB® and it is a function of the *Control System Toolbox*. The function header of the control parameter calculation and the implemented *lqr* MATLAB® function is shown below.

SYNTAX: *[Error, Error_Str] = CalculateControlParameter*

$$[\mathbf{K},\mathbf{P},\mathbf{e}] = lqr(\mathbf{A},\mathbf{b},\mathbf{Q},R)$$

By calling the *CalculateControlParameter* function, the control parameters of all dynos, which are available in the data storage, will be calculated. At the beginning of this function the matrices **A** and **b** (see equation 3.18) are created, as well as the matrix **Q** and the scalar R. Afterwards, the *lqr* MATLAB® function will be called and the correction control parameters K_p and K_i can be calculated by transforming the return parameter vector **K** (see equation 3.22). Then the parameter for the phase correction controller is defined by a tenth of K_i. Like the other observer parameters, the identified controller parameters are also saved in the data structure. For this reason, the *CalculateControlParameter* function only has the return parameters *Error* and *Error_Str* for the error handling.
The return parameter **P** of the *lqr*-function represents the **P** matrix, which solves the Riccati

equation (see formula 3.40). The return vector **e** includes the eigenvalues of the closed-loop system. However, these additional return parameters are not used anywhere.

5.1.6 Filter Parameter Function

The *CalculateFilterParameter* function is developed for determining the filter parameters of the dyno speed observer. As already explained in chapter 3.3, these parameters are calculated by the chirp test run. The quantities of the test run are saved in the program data structure, which is why the identification function only has the input parameter *WINDOW_SIZE*. The parameter *WINDOW_SIZE* defines the size of the window, which is used for calculating the FFT of the given speed and shaft torque signal. This parameter is specified in a power of two, by reason of the fast Fourier transformation algorithm construction, and it is set to 12 by default. This default value defines a window size of $N = 2^{12} = 4096$, which sets the frequency density around 1 Hz depending on the sampling rate of the controlling unit. The function call looks as follows:

SYNTAX: *[Error, Error_Str] = CalculateFilterParameter(WINDOW_SIZE)*

A frequency analysis of the measured speed and shaft torque signal is performed at the start of this identification function by the windowed fast Fourier transform algorithm (see chapter 3.3). The results are saved in the program's data storage for the visualisation plot in the identification software. Furthermore, these results are used to define the recommended filter parameters, which are also saved for the representation on the user interface. The return parameters *Error* and *Error_Str* are used to display some identification errors, which signalise that no filter parameter could be calculated.

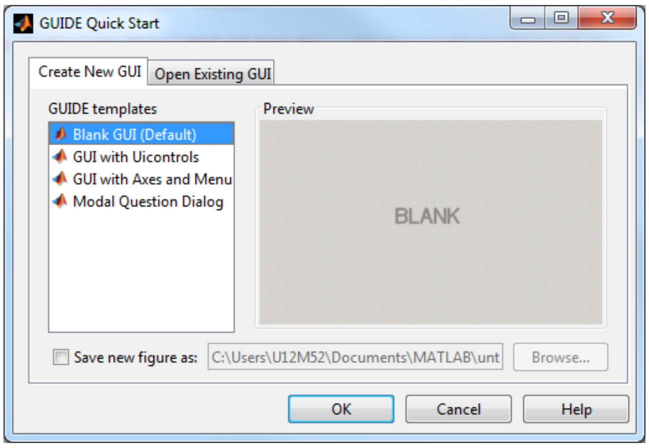
5.2 Identification Program

For a comfortable operation of the identification program *ObTune*, it is necessary to create a user-friendly graphical user interface. This enables easy handling of the deployed identification functions and the interaction with the program user. MATLAB® offers a tool for designing user interfaces for custom applications with the name *GUIDE (graphical user interface design environment)*.

5.2.1 MATLAB® GUIDE

The *GUIDE* tool can be started by typing the command *guide* at the MATLAB® prompt. After that, a *Quick Start* dialogue box, as shown in figure 5.2, opens. In this window the user can create a new GUI figure or open an existing GUI figure file. Furthermore, it is possible to open some prefabricated graphical user interface examples by choosing another template than the default one. If the default template is selected, the GUI designer environment will open with a blank background, which is illustrated by figure 5.3.
The predefined UI elements on the left side of the designer window can be added to the figure per drag and drop. The properties of the UI elements can be changed with an *Inspector* window, which opens by double-clicking on the UI element. During this designing process, the *GUIDE* tool automatically generates the MATLAB® code for constructing and using the UI in the background. The *GUIDE* tool therefore creates two different files. One *.fig* file, which stores

the layout and properties of the UI elements and one *.m* file that contains the program logic. In order to interact, it is important that these files have the same name.



Save new figure as: C:\Users\U12M52\Documents\MATLAB\unt

Figure 5.2: Start window of GUIDE.

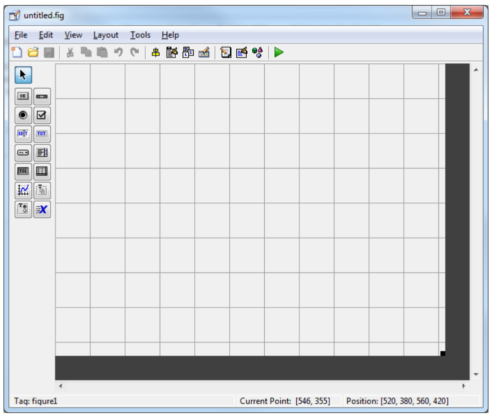

untitled.fig
File Edit View Layout Tools Help
Tag: figure1
Current Point: [546, 355]
Position: [520, 380, 560, 420]

Figure 5.3: Graphical user interface design environment tool.

It is also possible to create GUI elements such as user interface controls (buttons, sliders, etc.) or containers (panels, button groups, etc.) programmatically via MATLAB® commands. The advantage of creating the user interface by commands is that it offers more control over the design and it provides more flexibility regarding the software logic. For this reason, the main application of the identification software *ObTune* is developed via MATLAB® scripts and functions with UI commands. In contrast, simple subwindows, that are used from the main application, are designed by the *GUIDE* tool.

5.2.2 MATLAB® Compiler

With the MATLAB® Compiler, MATLAB® provides an opportunity to compile the created MATLAB® files into an executable application. Due to this, it is possible to share MATLAB® functions, scripts, etc. with someone who does not have a MATLAB® software from MathWorks®. The only thing which is necessary for the usage of the compiled executable is the MATLAB® *Runtime* environment on the workstation. This environment can be added to the executable or it can be downloaded during the installation process.
On all *PUMA* PC, the *MATLAB® Runtime* environment has already been installed, because the environment is also required for the automation and controlling software *PUMA*. The command below represents the usage of the *MATLAB Compiler*.

SYNTAX: *mcc [-options] .m-file [.m-file1 ... .m-fileN]*
[additional file1 ... fileN]

Building options as well as all needed application files are required by this *mcc* command function. The extent and meaning of the *mcc options* depend on the MATLAB® version and can be gathered from the proper MATLAB® help. In newer versions of MATLAB®, there is also a comfortable graphical user tool for creating an executable application. The prompt command *deploytool* triggers to start this graphical tool.
The identification method or rather the associated MATLAB® functions also use Simulink®

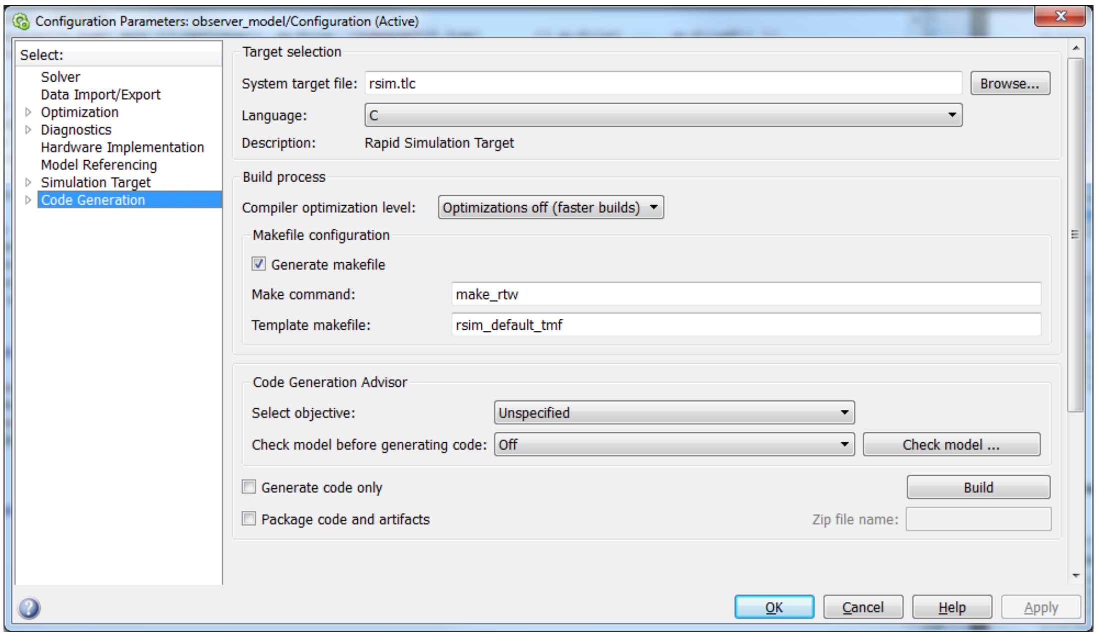
models (.mdl-files), but *MATLAB® Compiler* is not capable to compile Simulink® models for linking them to the resulting application. In this case, the Simulink® models need to be compiled to a usable executable first. This building process can be performed with the Simulink® *Coder*, which is also delivered by MATLAB®, and a common *C/C++ Compiler*.

5.2.3 Rapid Simulation Target

Simulink® provides the possibility to generate C/C++ code from the block diagram (.mdl-files). The code generation process can be triggered by the *Build Model* icon in the right corner of the Simulink® window. The process creates source code files in a subfolder of the current working directory. The settings for this generation process can be adjusted in the *Code Generation* tab of the *Model Configuration Parameters* window, which is shown in figure 5.4.
The first parameter of the *Code Generation* tab is for defining the *System target file* (see figure 5.4). The coder needs a system target files to translate the *.mdl*-files into source code and an executable. These target files define the system environment on which the source code will run. The *Simulink® Coder* already contains some predefined target files and it is possible to use third-party and custom target files as well.
Another important available setting of the *Code Generation* tab is the *Generate code only* checkbox. The building process will create code files only, if this checkable is selected. Otherwise, an executable application will also be produced by the preset *C/C++ Compiler* during the model building process. To set up the *C/C++ Compiler*, the following command need to be entered in the MATLAB® prompt:

SYNTAX: *mex -setup*

The generated code can be executed either with the command *!modelname* or *dos('modelname').* The execution with one of these commands creates a *.mat*-file. This file contains all quantities which will also be generated by the simulation process of the Simulink® model. Therefore, it is possible to run and plot the results of the Simulink® model without a *.mdl*-file.
For the constructed identification mechanism, it is essential that the parameters of the Simulink® model can be changed before the execution, because of the different dyno types and the optimization algorithm for the rise time parameter identification. This requirement can be satisfied by the *Rapid Simulation Target* (short *rsim*), which is already set up in figure 5.4. This *rsim* allows to change parameter values or input signals at the start of a simulation without a recompilation of the Simulink® model.
Such changeable parameters have to be declared as *tunable parameters* before compiling the Simulink® connection diagram. After performing the building process, the tunable parameters are accessible via a MATLAB® structure. This structure can be saved as a changeable *.mat* file, if everything has been configured correctly before. To ensure an automatic compilation of the whole identification application, the building process of Simulink® models is executed by a MATLAB® script.
Source listing 5.1 illustrates the section which ensures that the *System target file* of the current model is set to *Rapid Simulation Target.* The command *getActiveConfigSet* at line 3 returns an object, which enables the access to the *Model Configuration Parameters* for changing the target system.



Configuration Parameters: observer_model/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation

Target selection

System target file: rsim.tlc

Language: C

Description: Rapid Simulation Target

Build process

Compiler optimization level: Optimizations off (faster builds)

Makefile configuration

[✓] Generate makefile

Make command: make_rtw

Template makefile: rsim_default_tmf

Code Generation Advisor

Select objective: Unspecified

Check model before generating code: Off

[ ] Generate code only

[ ] Package code and artifacts

Zip file name:

Figure 5.4: Code generation window of a Simulink® model.

```
mdlName = 'modelname';
load\_system(mdlName);
cs = getActiveConfigSet(mdlName);
cs.switchTarget('rsim.tlc',[]);
save\_system(mdlName);
```

Listing 5.1: Set system target file.

As already mentioned, the *tunable parameters* must be defined before the building process. This definition operation is represented in source listing 5.2. Before the *tunable parameters* can be defined, the *RTWInlineParameters* checkbox needs to be must. Then the *tunable parameters* are able to be set via the command which is shown in line 2. The instructions below are necessary to specify the properties of the *tunable parameters*. The *TunableVarsStorageClass* property defines the storage class and thus the allocated memory for this parameter in the generated code. *TunableVarsTypeQualifier* represents the qualifier of the *tunable parameters* which can be defined as *const* for a non-modifiable or empty for a modifiable parameter.

```
1  set_param(mdlName,'RTWInlineParameters','on');
2  set_param(mdlName,'TunableVars',['J,x0,RisingTorqueRate,FallingTorqueRate,'...
3                                    'DelayTime,nNom,TNom,Kp,Ki,KiPhCorr']);
4  set_param(mdlName,'TunableVarsStorageClass',['Auto,Auto,Auto,Auto,Auto,Auto,'...
5                                                'Auto,Auto,Auto,Auto']);
6  set_param(mdlName,'TunableVarsTypeQualifier',',,,,,,,,,');
```

Listing 5.2: Define parameters as tunable.

If all these settings have been adjusted, the model can be built using the *rtwbuild* command. This coded instruction generates source files and builds them to an executable application. At the end of the building mechanism, the tunable parameter structure has been saved for the possibility of modification. This tunable parameter structure can be got by using the MATLAB® function *rsimgetrtp*. This procedure is shown in the code listing 5.3.

```
rtwbuild(mdlName);
rtp = rsimgetrtp(mdlName,'AddTunableParamInfo','on');
save('ParameterMatFile.mat','rtp');
```

Listing 5.3: Get and save tunable parameter structure

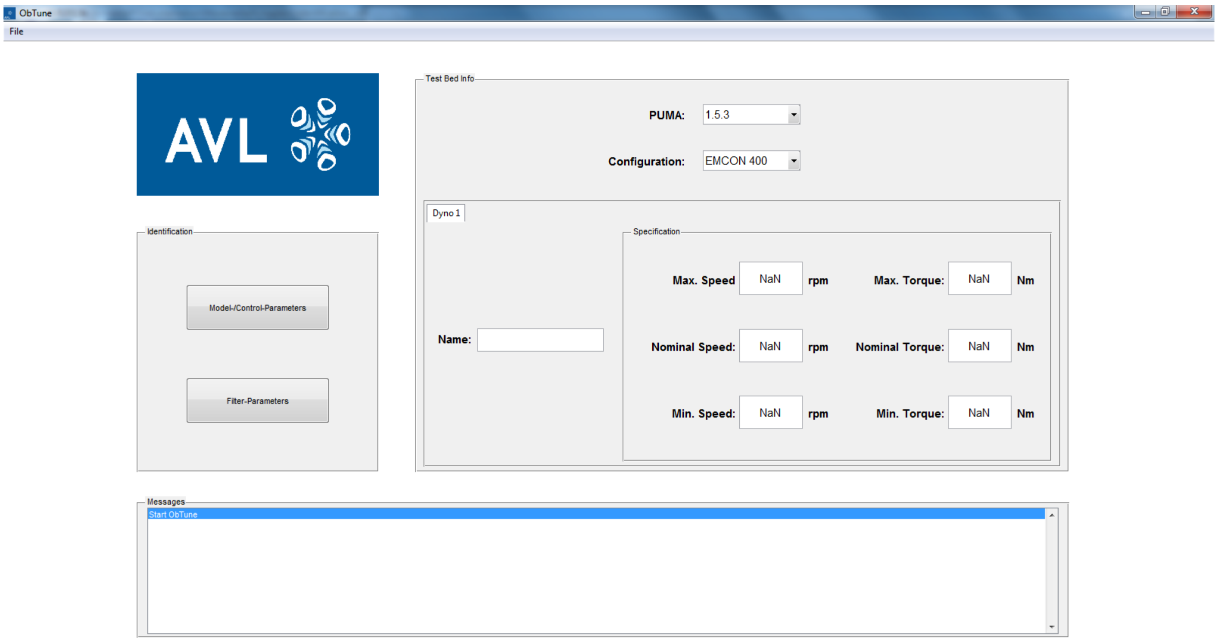
Listing 5.4 represents the modification of the tuning parameter and execution of the complied Simulink® model. The *rsimsetrtpparam* command takes the structure with tunable parameter information and sets the corresponding values. This parameter structure has to be saved before it can be used for the model executable. The execution instruction is shown in line 5 of listing 5.4. The additional parameter *-tf* defines the simulation time, *-p* specifies the structure of the tunable parameters and the input parameter *-o* sets the name of the output file, which contains the simulation results.
Thus, *Rapid Simulation Target* provides a comfortable way for modifying parameter values of a compiled Simulink® model without a recompilation, which is ideal for the identification software application.

```
1  rtp = rsimsetrtpparam(rtp,1,'J',J,'x0',x0,'RisingTorqueRate',RisingTorqueRate,...
2                        'FallingTorqueRate',FallingTorqueRate,...
3                        'DelayTime',DelayTime,'nNom',nNom,'TNom',TNom,'Kp',Kp,'Ki',Ki);
4
5  save('ParameterMatFile.mat','rtp');
6  dos('modelname.exe -tf 10 -p ParameterMatFile.mat@1 -o Result.mat')
```

Listing 5.4: Set tunable parameters and execute compiled model

5.2.4 Program Handling

If the user starts the identification software *ObTune*, the graphical interface, which is illustrated in figure 5.5, will appear. This operation window is clearly separated in three parts. The first part is named *Test Bed Info* (see figure 5.6) and it represents the important specifications for the identification mechanism of the automotive testbed. Before the identification can be started, the user need to enter the testbed configuration in this section like in figure 5.6.


ObTune
File

Identification
Model-/Control-Parameters
Filter-Parameters

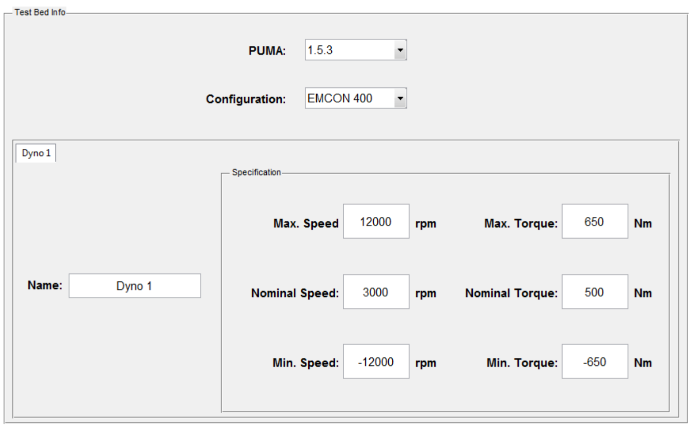
Test Bed Info
PUMA: 1.5.3
Configuration: EMCON 400

Dyno 1
Name:
Specification
Max. Speed NaN rpm Max. Torque: NaN Nm
Nominal Speed: NaN rpm Nominal Torque: NaN Nm
Min. Speed: NaN rpm Min. Torque: NaN Nm

Messages
Start ObTune

Figure 5.5: Main menu window of the identification software.

The next part is the *Identification* section that includes the selection buttons for the different identification operations. Clicking on the *Model-/Control-Parameters* button opens the graphical user interface for the identification of the observer model and controller parameters. Clicking on the *Filter-Parameters* button opens the window for the identification mechanism of the filter parameters. If one of these buttons is pressed and something is not entered correctly in the *Test Bed Info* section, for example the maximum speed is lower than the nominal dyno speed, an error message will be appear in the message box, which presents the third part of the main window.


Test Bed Info
PUMA: 1.5.3
Configuration: EMCON 400

Dyno 1
Name: Dyno 1
Specification
Max. Speed 12000 rpm Max. Torque: 650 Nm
Nominal Speed: 3000 rpm Nominal Torque: 500 Nm
Min. Speed: -12000 rpm Min. Torque: -650 Nm

Figure 5.6: Start window of GUIDE.

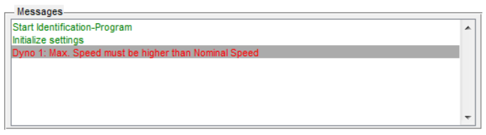
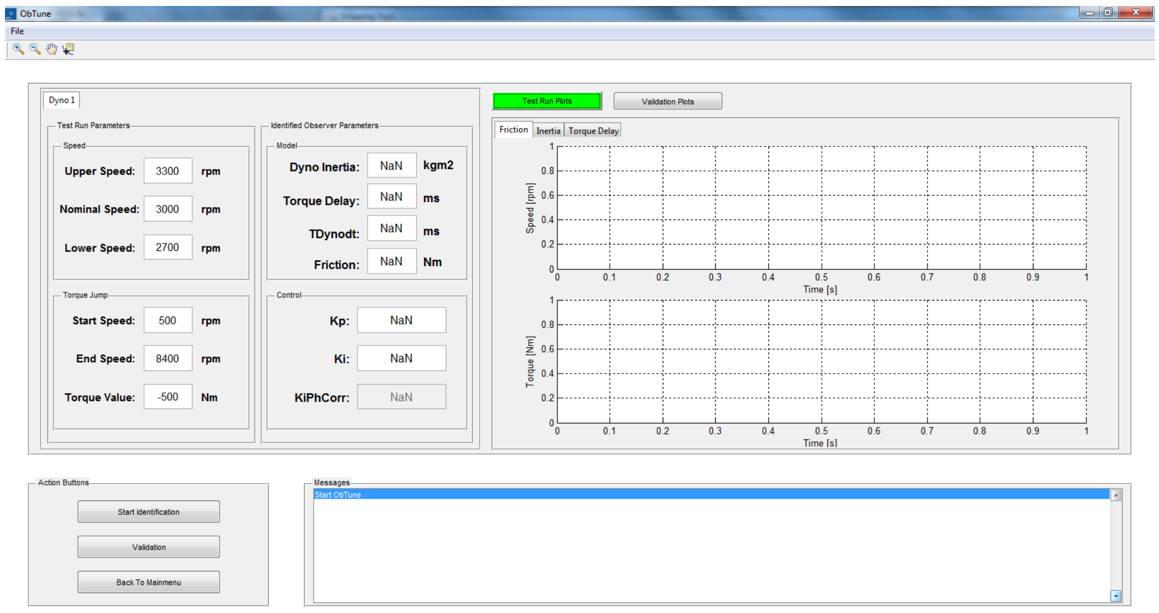

Messages
Start Identification-Program
Initialize settings
Dyno 1: Max. Speed must be higher than Nominal Speed

Figure 5.7: Graphical user interface design environment tool.

This message box is available in every interaction window of the application and shows all occurring messages. Error messages are coloured in red, successful events are represented in green and warnings or standard information messages are black. Only if everything has been entered correctly in the *Test Bed Info* section, the identification process can be started by clicking on one of the identification buttons.

5.2.4.1 Identification of Model and Control Parameters

When the identification of the model and control parameters is started with the attendant button, the user interface in figure 5.8 will appear for the software user. Like the main software window, the user interface is separated into several sections.



Test Run Parameters

Speed

Upper Speed:	3300	rpm
Nominal Speed:	3000	rpm
Lower Speed:	2700	rpm

Torque Jump

Start Speed:	500	rpm
End Speed:	8400	rpm
Torque Value:	-500	Nm

Identified Observer Parameters

Model

Dyno Inertia:	NaN	kgm2
Torque Delay:	NaN	ms
TDynodt:	NaN	ms
Friction:	NaN	Nm

Control

Kp:	NaN
Ki:	NaN
KiPhCorr:	NaN

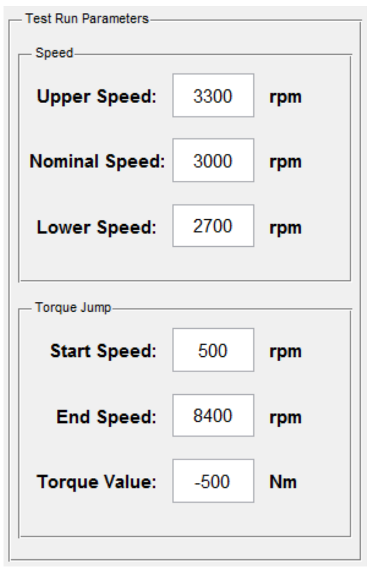
Action Buttons

Messages

Start ObTune

Figure 5.8: User interface for identification of the model and control parameters.

At the beginning of the identification procedure, the *Test Run Parameters* section, which is shown in figure 5.9, is the most interesting part. In this panel, the test run parameters for the dialogue window of the BSQ testbed test run (see figure 4.13) is presented to the user. These parameter values are calculated on the basis of the *Test Bed Info* data. However, the user already has the option of changing the recommended *Test Run Parameters* according to their discretion. The only important thing is that these parameter values correlate with the entered values in the test run dialogue window.
As already described in chapter 4.2 the model and control test run creates two recording files for the identification process. These two files are now required in the identification software. The identification sequence starts by pressing the *Start Identification* button in the *Action Button* panel. Thus a dialogue window pops up (see figure 5.11), in which the test run recording files have to be selected for the identification process. If the recording files are selected, the process will start after confirming with the OK button in the same window.



Test Run Parameters

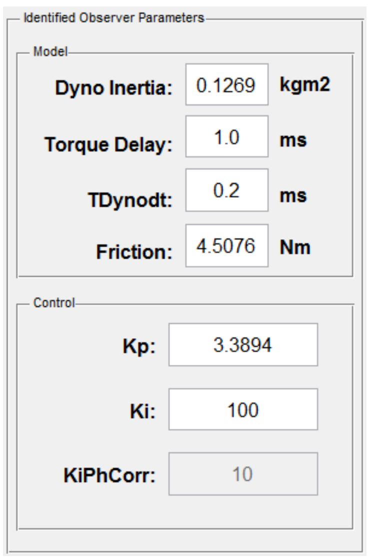
Speed

Upper Speed:	3300	rpm
Nominal Speed:	3000	rpm
Lower Speed:	2700	rpm

Torque Jump

Start Speed:	500	rpm
End Speed:	8400	rpm
Torque Value:	-500	Nm

Figure 5.9: Recommended test run parameters.



Identified Observer Parameters

Model

Dyno Inertia:	0.1269	kgm2
Torque Delay:	1.0	ms
TDynodt:	0.2	ms
Friction:	4.5076	Nm

Control

Kp:	3.3894
Ki:	100
KiPhCorr:	10

Figure 5.10: Calculated observer parameters.

During the identification, a process bar is displayed for visualising the remaining process time. After the identification procedure, the process bar closes, a message in the message box appears and the calculated observer parameters are shown in the *Identified Observer Parameters* panel, which is presented in figure 5.10. Furthermore, test run and validation plots, which are shown in figures 5.12 and 5.13, are displayed in the graphical user interface.

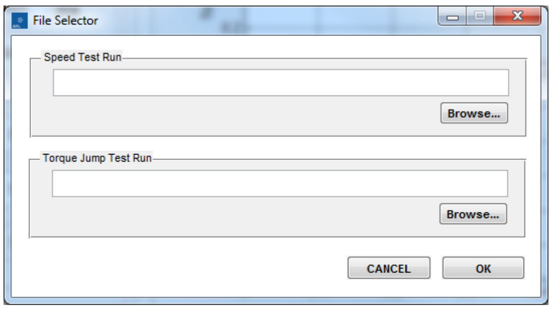


Figure 5.11: Dialogue window for selecting the test run recording files.

The user has the opportunity of switching between test run and validation plots by pressing the corresponding buttons. Executed test runs as well as the detected calculation points for the identified model parameters are plotted in the test run section. Essentially, the plot in figure 3.3 is shown in the *Friction* tab, figure 3.4 is displayed in the *Inertia* tab and the plot 5.1 is represented in the *Torque Delay* figure tab. These plots give the software user the ability to control the identification mechanism and therefore the accuracy of the detected model parameters.
As the name already suggests, the validation plots contain figures which give a visual representation of the identified parameter's quality and the behaviour of the correction controller as well as the whole speed observer.

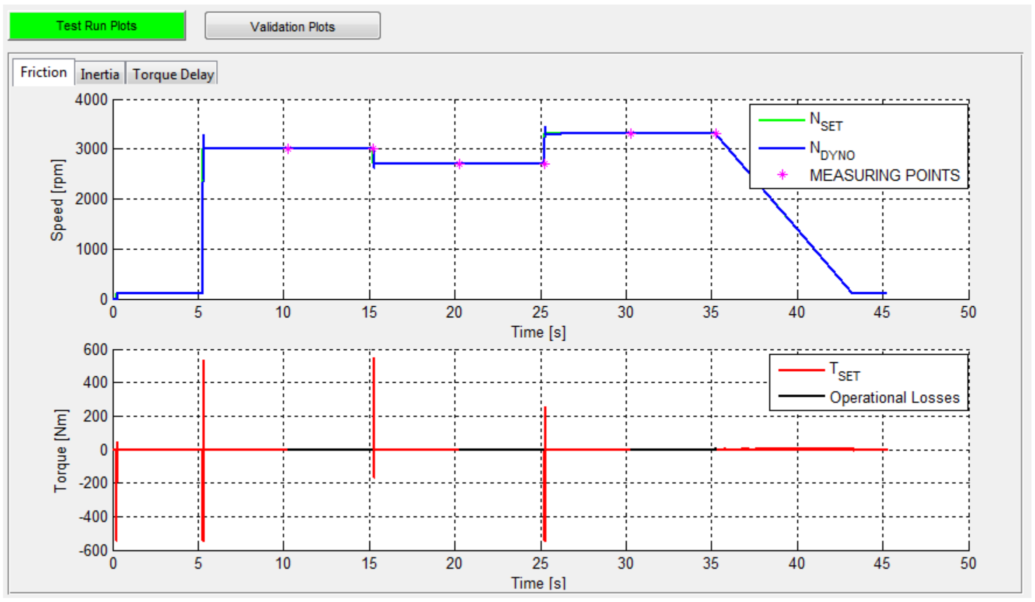


Figure 5.12: Plots of the identification test runs

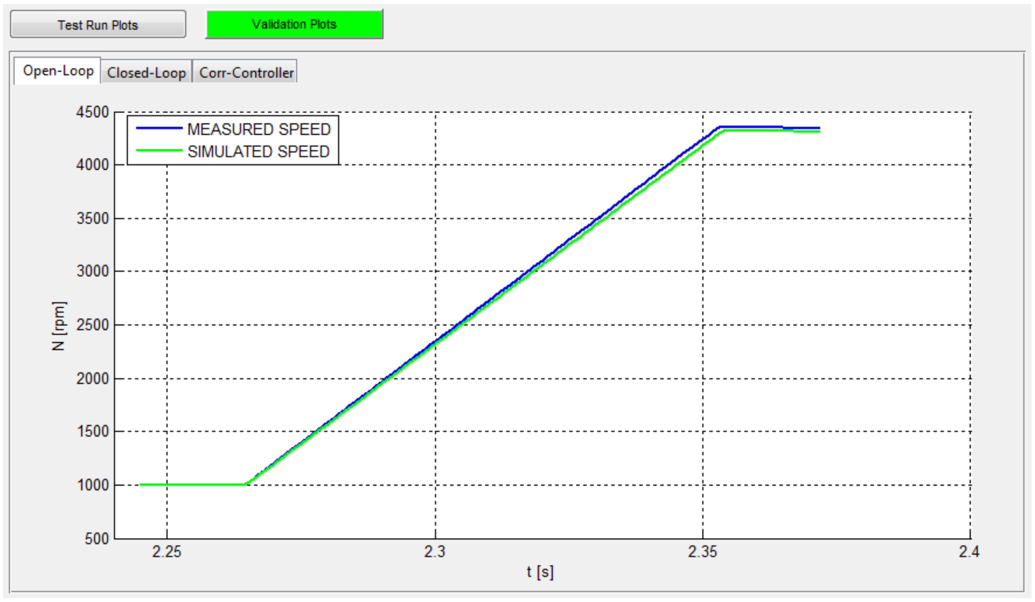


Figure 5.13: Plots of the parameters validations.

The first plot in the *Open-Loop* tab represents the speed behaviour of the testbed dyno and the observer model with the identified parameters. A good match between these two signals verifies a good determination of the model parameters. This verification is performed with the aid of a *Rapid Simulation Target* compiled Simulink® block diagram of the observer dyno model. This Simulink® model is shown in figure 5.14. The simulation model parameters (Inertia, Torque Delay, Rise Time) and the integrator initial value *x0* are declared as tunable parameters, because they depend on the given dyno and performed test run. The initial condition of the integrator describes the dyno start speed at the beginning of the test run.
The input value of the simulation model is applied by a *.mat* file, which contains the recorded torque set signal of the torque jump test run. The simulation output, which represents the

simulated speed signal of the dyno model, will be saved in the data storage of the identification program for plotting the *Open-Loop* validation figure.
Figure 5.13 shows the *Open-Loop* validation of the engine testbed identification. The blue line represents the recorded, and the green the simulated speed signal with the model parameters of figure 5.10. The fact that the two lines match, illustrates excellently that the identified model parameters simulate the real dyno behaviour very well. Therefore, it can be assumed that the identification procedure of the model parameters was successful.

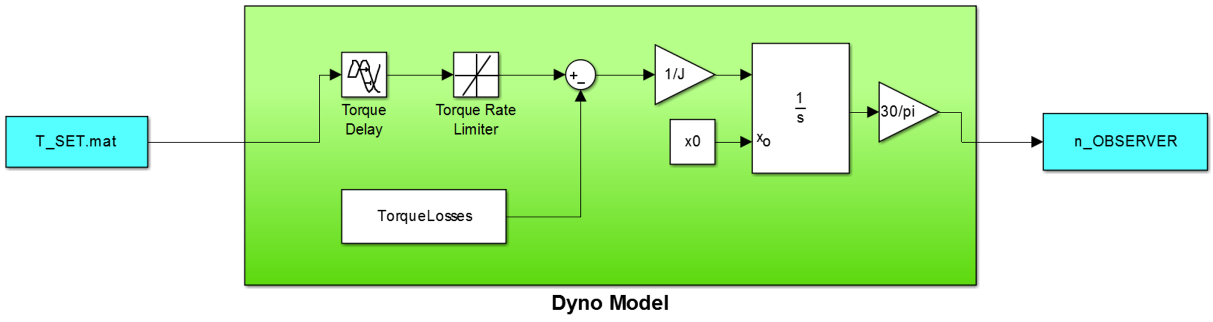


Figure 5.14: Simulink® dyno model for Open-Loop validation.

The other two validation plot tabs *Closed-Loop* and *Corr-Controller* are created by the compiled Simulink® model which is shown in figure 5.15. This simulation model represents the behaviour of the speed observer with the implemented correction controller. The controller parameters from the *Control* panel of the user interface (see figure 5.10) are used for this validation procedure. The *Closed-Loop* tab shows the measured and simulated speed signal just as the *Open-Loop* tab, but with an active correction controller. This illustrates how the observer model works with the identified correction controller parameters. The last plot tab gives the user a feedback of the controller output. The *Corr-Controller* tab represents the output of the correction controller. The output of the phase correction controller is not illustrated, because there is no additional benefit. As illustrated by figure 5.15, the output of the phase correction controller has no influence of the controlling behaviour. This is why the control parameter in the *Identified Observer Parameters* panel (see figure 5.10) is not editable. This *KiPhCorr* parameter is always set to one-tenth of the *Ki* parameter of the correction controller. Generally, it must be noted that the correction controller output will not match with the real correction controller output of the automotive testbed. There are some additional controllers at the testbed which have an influence in the controlling behaviour and therefore also have an influence on the correction controller of the speed observer.
As for the model parameter validation, all observer parameters are declared as tunable and the recorded torque and speed signal of the torque jump test is used for the simulation input *.mat*-file. The outputs *n_OBSERVER* and *CorrController_OUT* are stored in the data structure for plotting the described validation plots.

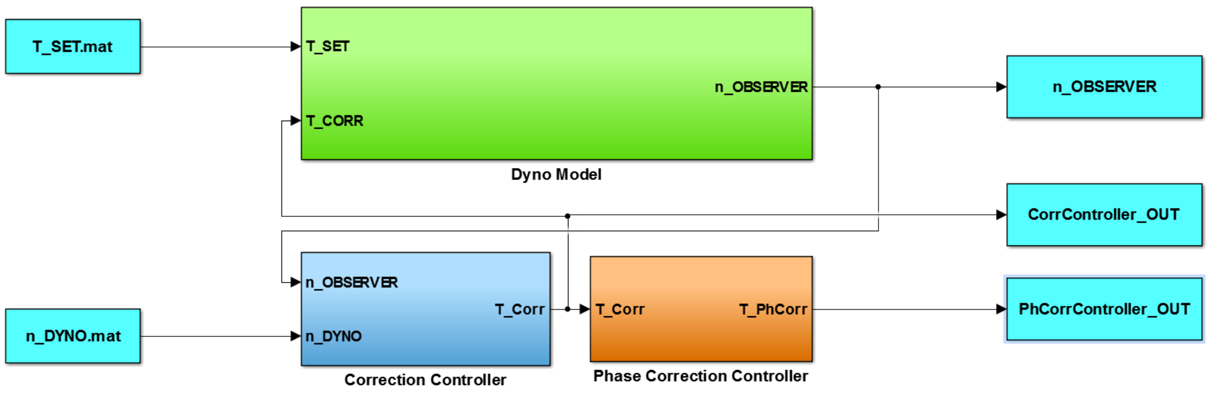
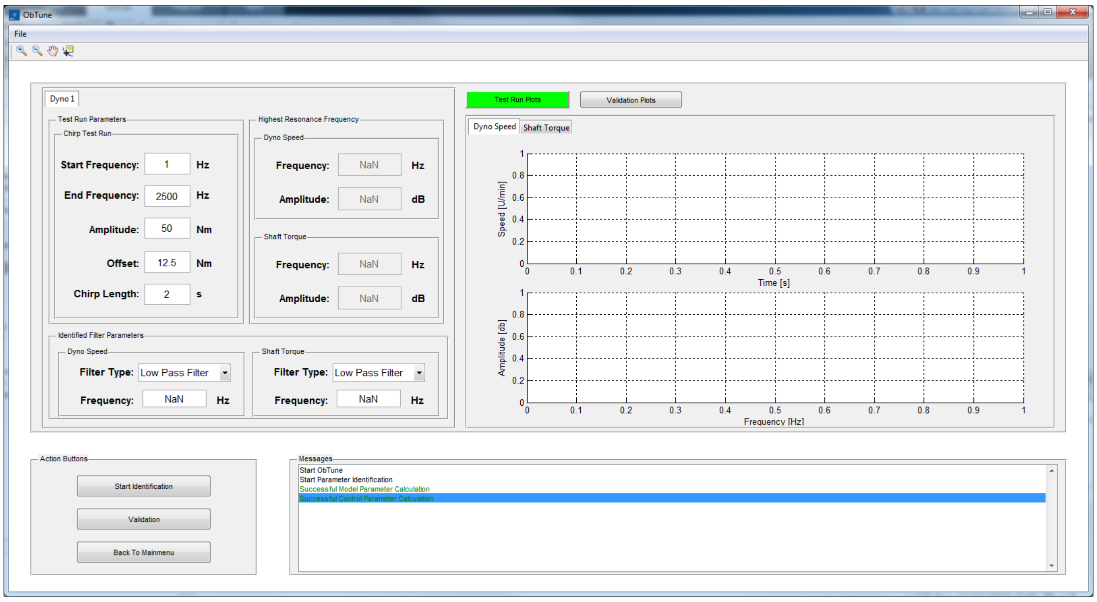


Figure 5.15: Simulink® dyno model for Closed-Loop and controller validation.

However, it is still possible that an error occurs during the identification procedure and the observer parameter calculation cannot be finished successfully. Another possibility is that the detected dyno speed observer parameters misfit for some reason and the validation plots signalise that the real and simulated dyno speed signals do not accord well.
For these undesirable but possible identification scenarios, a very helpful software functionality is implemented, which can be operated by pressing the *Validation* button of the *Action Button* panel. This software button induces a selective execution of the validation process. Due to the fact that the observer parameters of the *Identified Observer Parameters* panel are editable, the software user has the option of approximating the ideal observer parameters in an iterative way by changing the parameters in the *Identified Observer Parameters* panel.
Firstly, the model parameters should be changed as long as the signals of the *Open-Loop* validation plot correspond well. After that, the control parameters should be adapted as desired. The validation process can be operated only if an identification was performed before, because the test run data are necessary for the Simulink® models that are used for the validation procedure. If the *Validation* button is pressed before test run records for the *ObTune* software are available, the validation aborts with an error message.
At the end, the calculated model and control parameters, which are found by the automatic tuning algorithm or with manual tuning, should be entered in the corresponding settings file of the *PUMA Open* automation software for using it at the automotive testbed. For continuing the observer parameter identification, the last button *Back To Mainmenu* returns the user to the main function window.

5.2.4.2 Identification of Filter Parameters

By starting the filter parameters identification, the graphical user interface, which is shown in figure 5.16 is presented for the user. Like the interface for the model and control parameters identification, the filter parameter calculation window is also separated in individual sections and generally works according to the same principle.



Dyno 1

Test Run Parameters — Chirp Test Run

Start Frequency:	1	Hz
End Frequency:	2500	Hz
Amplitude:	50	Nm
Offset:	12.5	Nm
Chirp Length:	2	s

Highest Resonance Frequency

Dyno Speed

Frequency:	NaN	Hz
Amplitude:	NaN	dB

Shaft Torque

Frequency:	NaN	Hz
Amplitude:	NaN	dB

Identified Filter Parameters

Dyno Speed

Filter Type:	Low Pass Filter	
Frequency:	NaN	Hz

Shaft Torque

Filter Type:	Low Pass Filter	
Frequency:	NaN	Hz

Messages

```
Start ObTune
Start Parameter Identification
Successful Model Parameter Calculation
Successful Control Parameter Calculation
```

Figure 5.16: Graphical user interface for filter parameter identification.

As discussed in chapter 4.2, the chirp test run also needs some scaling execution parameters. A recommendation for these test run parameters is given in the parameter panel, which is is shown in figure 5.18. These parameters are editable and can be adapted by the user as necessary. The parameter panel also contains two further sections that are filled after the identification process. First, the panel *Highest Resonance Frequency* displays the frequencies with the maximum amplitude of the speed and shaft torque signal of the chirp test run. This panel only represents the measurement of the largest disturbance frequencies and, therefore, the parameters are not editable in this part. The third section illustrates the identified filter parameters for the speed and shaft torque filter.

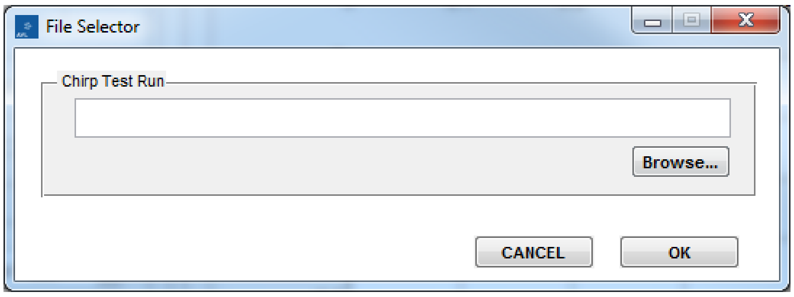
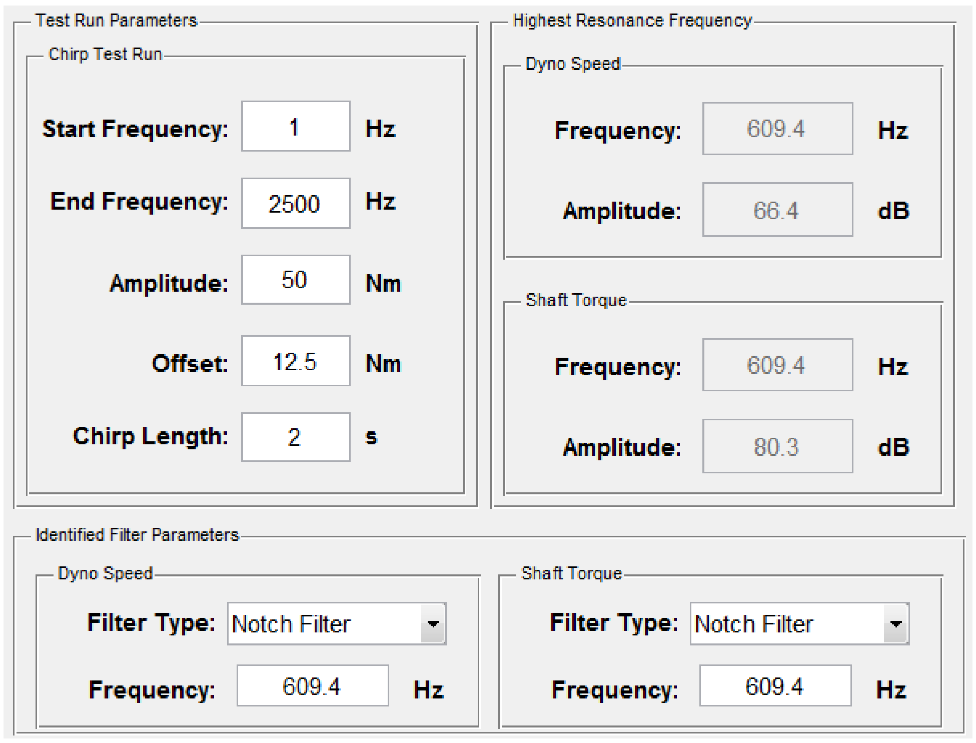


Figure 5.17: Dialogue window for chirp test selection.

If the identification mechanism is started by pressing the *Start Identification* button, a dialogue window, just as for the model and control parameters before, opens for selecting the chirp test run recording file (see figure 5.17). After selecting the file by using the *Browse* button, the identification process can be started by confirming with the *OK* button. In contrast, if the user wishes to abort the identification mechanism, they have the option to close the dialogue section window by pressing the ESC key or the *Cancel* button. During the identification, a waiting bar is represented for illustrating the actual progress.



Test Run Parameters

Chirp Test Run

Start Frequency:	1	Hz
End Frequency:	2500	Hz
Amplitude:	50	Nm
Offset:	12.5	Nm
Chirp Length:	2	s

Highest Resonance Frequency

Dyno Speed

Frequency:	609.4	Hz
Amplitude:	66.4	dB

Shaft Torque

Frequency:	609.4	Hz
Amplitude:	80.3	dB

Identified Filter Parameters

Dyno Speed

Filter Type:	Notch Filter	
Frequency:	609.4	Hz

Shaft Torque

Filter Type:	Notch Filter	
Frequency:	609.4	Hz

Figure 5.18: Parameter panel of the filter identification window.

After identification, the determined filter parameters, the highest resonance frequencies with their amplitude as well as the test run and validation plots are shown in the appropriate illustration blocks (see figure 5.18). If an error occurs during the identification, a corresponding message will be written in the message box. In this case, those sections, which are affected by the identification error, stay unchanged. If for example no filter frequency parameter can be calculated, the appropriate text box will still contain the value *NaN*. The *Test Run Plot* section (see figure 5.19) illustrates the recorded speed and shaft torque signal during the chirp test and their fast Fourier transformation. At the *Validation* plot part (see figure 5.20), the filtered speed and shaft torque signal and the corresponding fast Fourier transformation is represented.
To still perform a successful identification despite of an error, the validation functionality is also given in the filter parameter identification window. The same validation principle as described before in the model and control identification window is used. By clicking on the *Validation* button, the editable parameters of the *Identified Filter Parameters* panel are consulted for fitting the Simulink® validation model, which is shown in figure 5.21.

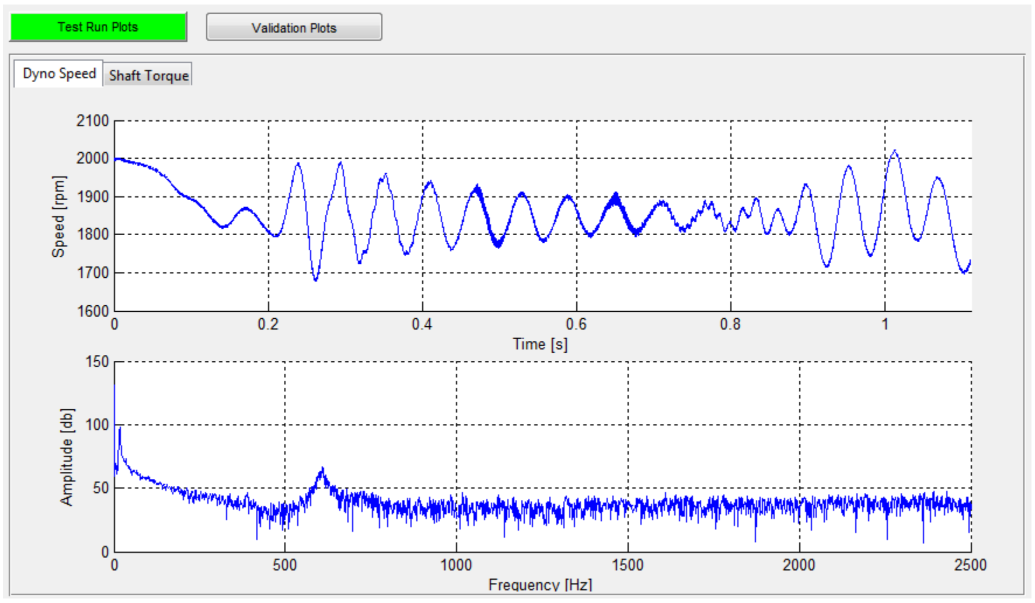


Figure 5.19: Plots of the chirp test runs

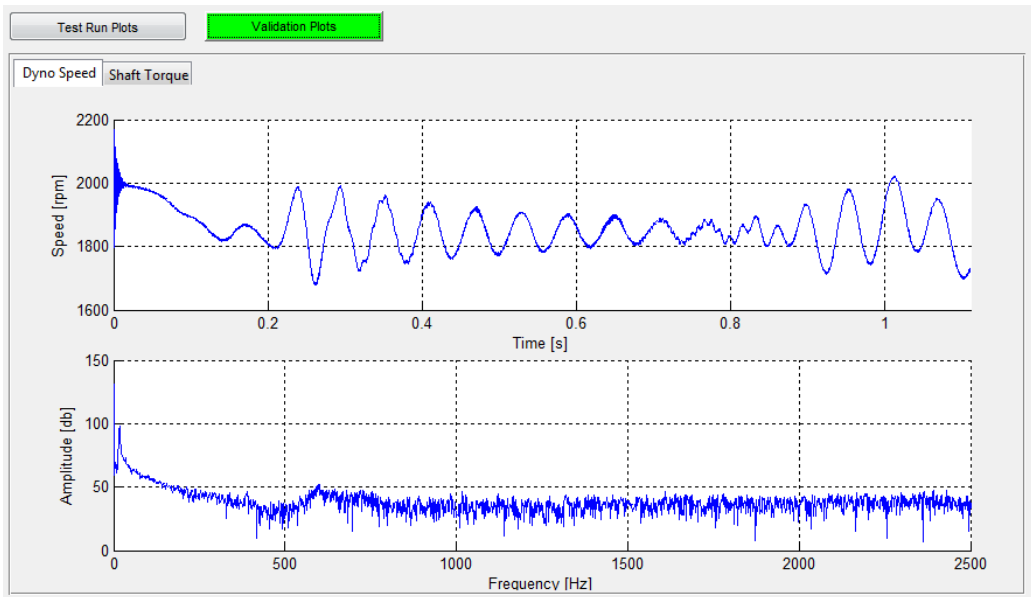


Figure 5.20: Plots of the filter parameters validations.

The validation model at figure 5.20 represents the implemented speed and shaft torque filter of the dyno speed observer. At the validation, the filter will be configured and then the measured testbed signal is passed through. The received filtered signals and their FFT constitute the validation plots at the graphical user interface.
Figure 5.19 shows the recorded speed signal of the chirp test run on the engine testbed. The corresponding validation plot is represented below in Figure 5.20. It shows that the identified filter parameters, which are visible in figure 5.18, eliminate the resonance frequency peak successfully. The improvement of the speed signal caused by the filter can also be recognized in figure 5.20.
As described in chapter 3.3 the implemented filter is a second order digital *IIR* filter. Simulink®

already provides an implemented filter block with adjustable numerator and denominator for defining the filter characteristic, but that block is not used for the dyno observer of the *EMCON* software. The reason for this is that after the model compilation it is not possible to change the filter parameters in a correct way. Due to this, the *IIR* filter is realized with tunable gain blocks for changing the characteristic. The gain coefficients can be declared as tunable parameters. The values of the gain blocks are calculated by an AVL MATLAB® script. This script needs the filter type (low-pass or notch filter) and the filter frequency (cut-off or trap frequency) as input parameters. The return parameters are the gain values *a1, a2, b1, b2* and *b3* of the implemented filter (see figure 5.21). Therefore, this implementation also has the advantage that it can easily be used for the validation mechanism of the *ObTune* software.

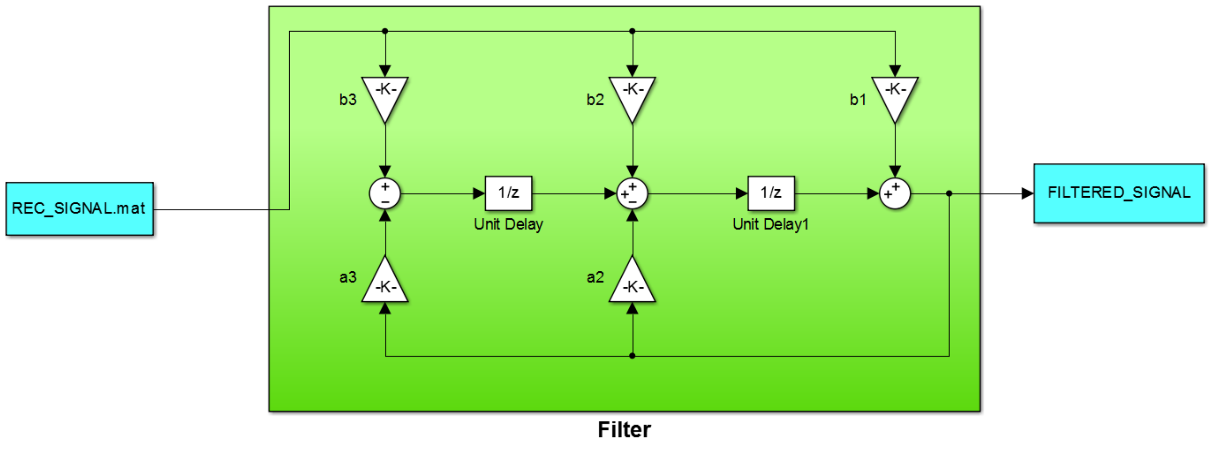


Figure 5.21: Simulink® dyno model for filter parameter validation.

Bibliography

[1] Anne Angermann, Michael Beuschel, and Ulrich Wohlfarth Martin Rau. *MATLAB - Simulink - Stateflow: Grundlagen, Toolboxen, Beispiele*. Walter de Gruyter, 2014.

[2] Andreas Antoniou and Wu-Sheng Lu. *Practical Optimization: Algorithms and Engineering Applications*. Springer Science and Business Media, 2007.

[3] Paul Dobrinski, Gunter Krakau, and Anselm Vogel. *Physik für Ingenieure*. Springer-Verlag, 2013.

[4] Neil A. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press, 1999.

[5] Steffen Goebbels and Stefan Ritter. *Mathematik verstehen und anwenden – von den Grundlagen bis zu Fourier-Reihen und Laplace-Transformation*. Springer-Verlag, 2013.

[6] Thomas Haller and Harald L. Schedl. *Digitale Signalverarbeitung: Grundlagen, Theorie, Anwendungen in der Automatisierungstechnik*. Springer-Verlag, 2013.

[7] Norbert Herrmann. *Höhere Mathematik: für Ingenieure, Physiker und Mathematiker*. Oldenbourg Verlag, 2007.

[8] M. Horn and J. Zehetner. A brake-testbench for research and education. In *Control Applications, 2007. CCA 2007. IEEE International Conference on*, pages 444–448, Oct 2007.

[9] Martin Horn and Nicolaos Dourdoumas. *Regelungstechnik*. Pearson Studium, 2004.

[10] Karl-Dirk Kammeyer, Kroschel Kristian, Armin Dekorsy, Dieter Boss, and Jürgen Rinas. *Digitale Signalverarbeitung: Filterung und Spektralanalyse mit MATLAB-Übungen*. Springer-Verlag, 2013.

[11] Fritz Kurt Kneubühl and Damien Philippe Scherrer. *Lineare und nichtlineare Schwingungen und Wellen*. Springer-Verlag, 2013.

[12] Bela G. Liptak. *Optimization of Industrial Unit Processes, Second Edition*. CRC Press, 1998.

[13] Jan Lunze. *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung*. 8. Auflage. Springer-Verlag, 2014.

[14] André Neubauer. *DFT - Diskrete Fourier-Transformation: Elementare Einführung*. Springer-Verlag, 2012.

[15] Eckhard Spring. *Elektrische Maschinen: Eine Einführung*. Springer-Verlag, 2013.

Table of Figures

Table of Tables

Table of Listings