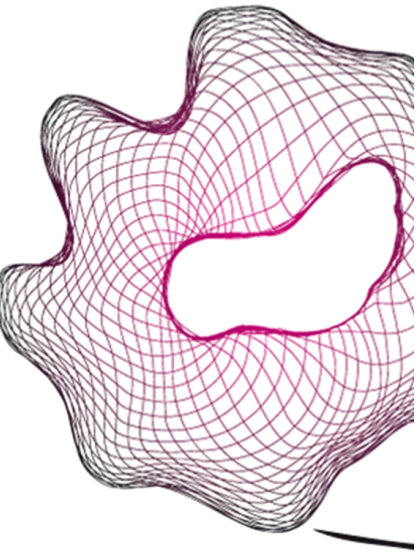
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science



Selective Harmonic Elimination and Model Predictive Control in Motor Drives Application for Electromagnetic Excited Noise Mitigation

M. J. Grootte Bromhaar

MSc. Thesis

October 2024

Supervisors:
University Supervisor: T. Batista Soeiro
Daily supervisor: A. Iyer

Power Electronics Group
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

This report explores various techniques for controlling motor drives and it also delves deep into harmonic elimination techniques. Traditional motor controller methods like Field Oriented Control (FOC) and Direct Torque Control (DTC) were examined. The FOC method's performance is limited by the bandwidth of the PI controllers. This causes slower convergence to target setpoints. Opposed to that, the incorporated indirect Model Predictive Control (MPC) approach provided significantly faster convergence than the PI controllers. It did struggle with exact reference tracking due to limitations in its interaction with the Space Vector Pulse Width Modulation (SPWM) block.

Additionally, this report includes various modulation techniques, like SPWM for FOC. Issues arose with the Equal Area Modulator, which is part of one of the implemented SHE algorithms. To try and solve these issues a Phase Locked Loop (PLL) was designed, which successfully filters away unwanted noise coming from the PI outputs or changing motor speed, but the quality of the current waveforms did not improve unfortunately. The issues could also be there due to inaccuracies in timing among the gating signals or plant behavior needs to be taken into account. However, the Adaptive Selective Harmonic Elimination (ASHE) method, based on the Least Mean Squares (LMS) algorithm, demonstrated promising results by reducing harmonics and maintaining stable (dynamic) control of the inverter outputs.

The indirect MPC method showed more harmonic reduction when compared to the LMS-based SHE method connected to the PI-based FOC controller. This scheme reduces Total Harmonic Distortion (THD) the best as well. Furthermore, the MPC's ability to quickly adapt to dynamic conditions makes it an attractive solution for motor drive applications.

All of these (motor) controllers were implemented in PLECS, where an example simulation model provided an excellent start. The (A)SHE methods and indirect MPC method were implemented in C-script code blocks, which was very fitting, as for these methods to work on hardware, a digital implementation in code is needed.

Building upon the great results of purely the MPC method, by combining MPC with LMS-based SHE, could offer significant potential for harmonic elimination and dynamic control in motor drives. Future work can focus on enhancing this motor controller scheme by filters, observers or integrating the LMS compensation in a different place to further improve system performance. Implementing the compensation in different reference frames (like dq or $\alpha\beta$) could make use of the speed of the MPC to optimize harmonic compensation.

Contents

Chapter 1

Introduction

In today's world, technological development focuses on creating more efficient and sustainable energy sources. The search for efficiency and sustainability is present in many industries, such as in industrial settings, consumer electronics, the automotive scene or other forms of electrical transportation. The growing population further enhances the need for optimising energy usage, to reduce the environmental impact of our actions.

Electrification is important in this context, as this pushes the norm away from fossil-fuel based systems, which leads to possibly cleaner energy usage, but surely more efficient energy usage.

A necessary component for electrifying technologies is the electric motor, which is found in so many products and are used for so many applications that coming up with examples is trivial. However, for these motors to operate to certain demands, a fitting control system must be put into place. This is where electric motor drives are called upon.

In modern electric motor drive applications, mitigating electromagnetic noise and vibration ensures a comfortable user experience across a wide range of speeds. This is particularly advantageous in Electric Vehicles (EV's) or maritime applications. Traditional approaches to reducing this noise and vibration have relied on auxiliary methods such as attenuation or absorption, which can increase overall costs and system complexity [2]–[4]. An alternative solution involves active control techniques that directly address the sources of noise and vibration within the motor drives.

This master's thesis focuses on the study and/or implementation of Selective Harmonic Elimination (SHE) and Model Predictive Control (MPC) to improve the performance of motor drives. SHE is a modulation strategy designed to eliminate specific harmonics in the inverter output, thereby reducing unwanted electromagnetic emissions. By developing and simulating SHE models on a motor consisting of a resistive-inductive (RL) load, we aim to validate the effectiveness of this technique in practical scenarios. In this thesis a Permanent Magnet Synchronous Machine (PMSM) is used to which the aforementioned techniques will be applied.

Additionally, the integration of MPC provides a robust framework for optimizing the dynamic response of motor drives. MPC is well-suited for applications requiring precise control over system performance metrics, such as harmonic reduction, acoustic noise minimization,

and common-mode voltage suppression [1]. This approach not only enhances the efficiency and reliability of motor drives but also addresses safety concerns associated with common-mode current circulation.

The scope of this research includes a comprehensive literature review on predictive control and optimum pulse pattern modulation in motor drives, analytical modeling of voltage source inverters (VSIs), and benchmarking against conventional modulation techniques such as Sinusoidal Pulse Width Modulation (SPWM). The ultimate goal is to implement and validate the combined SHE and MPC strategies using a simulation platform, which in this case is PLECS.

To reach the aforementioned goal, a fitting research question needs to be formulated. Taking the contents of this report into account and its scope, the research question to be answered will be:

1. What techniques are used in the control and modulation of electric motor drives?
2. How can MPC and SHE techniques be integrated in a motor drive, where unwanted harmonics need to be eliminated?
3. How do MPC and certain SHE techniques affect the dynamic performance of motor drives?

In this thesis, the aim is to demonstrate the potential of integrating SHE and MPC to achieve superior performance in motor drive applications, thereby contributing to the development of possibly more efficient and quieter electric machines.

Chapter 2

Relevant background information

2.1 Operation of a permanent magnet machine

A rudimentary permanent magnet machine is pictured in Fig. 2.1. This specific one is used solely as a tool to comprehend the operation of a machine and is not used in the implementation. The motor consists of a rotor and stator, where the stator is the outer part

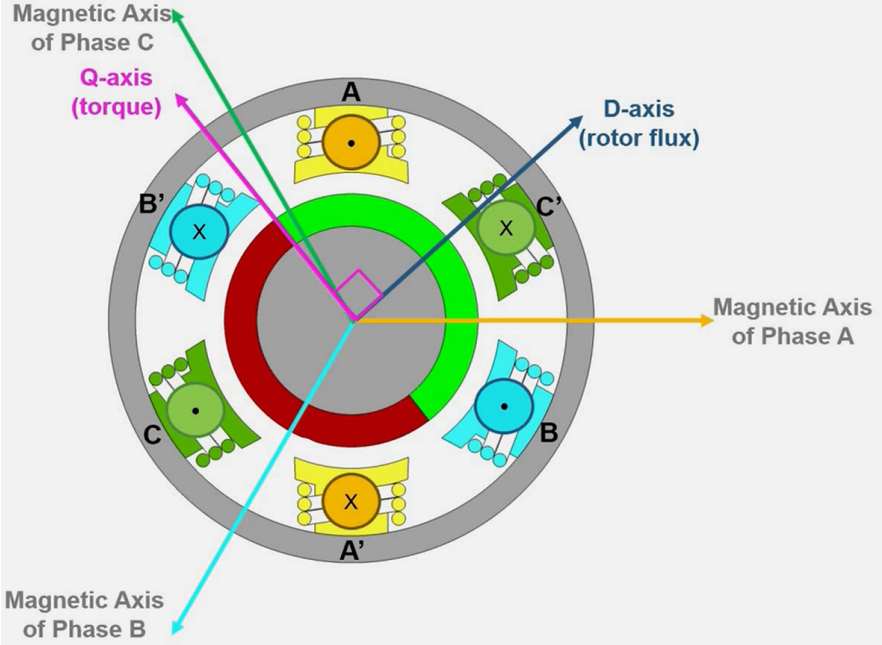


Figure 2.1: PMSM machine, where the red and green parts of the rotor represent the magnetic north and south pole [5]

of the motor, containing the coils. The rotor is the inner part in this specific motor, which turns. The rotor consists of one pole pair of permanent magnets, which means that every mechanical revolution, the stator completes an electrical revolution as well.

In this report, a permanent magnet machine will be used which has $L_d = L_q$, equivalent to a saliency ratio (L_q/L_d) of 1. To make the rotor of the machine turn, the right coils of the three phases should be excited appropriately. In the example in Fig. 2.1, to make the motor

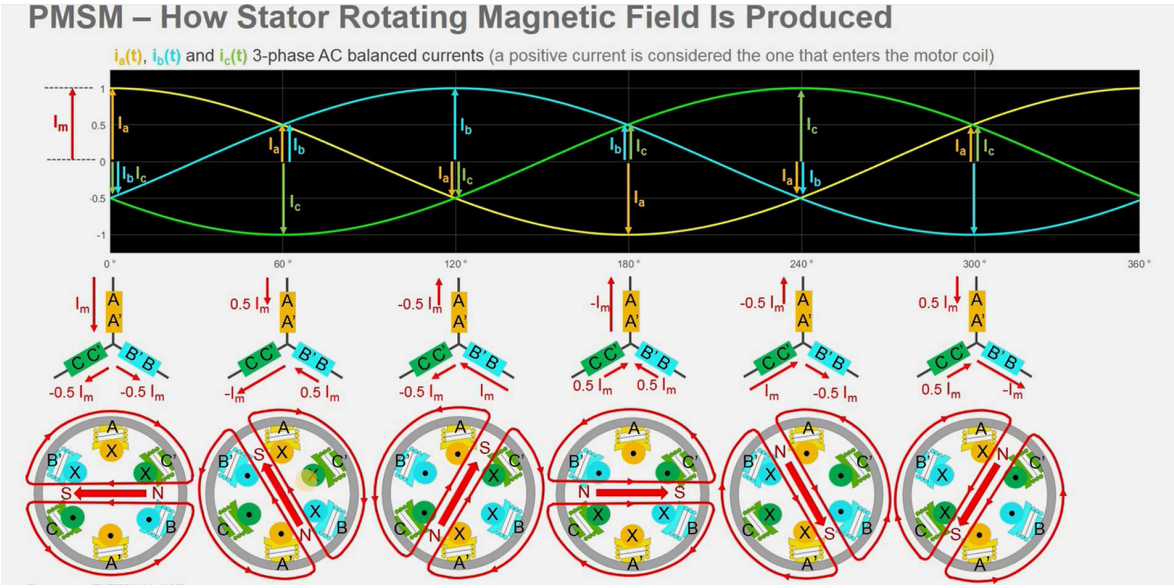


Figure 2.2: Depiction of the stator commutation of a PMSM [5]

turn clockwise, coil A and A' should produce the "strongest" South and North pole to the side of the coil that is closest to the rotor, when the Q-axis of the rotor is orthogonal to the magnetic axis of phase A, resulting in maximum torque and zero current on the Direct (D) axis. As there are no magnetic mono-poles, the outer sides of the coils will of course produce the complementary magnetic pole. In this case, the excitation of coils C and C' should be practically zero, as non-zero D current would result otherwise, which means that there is a magneto motive force acting through the center of the motor, which results in energy loss. In Fig. 2.2 the commutation of the stator can be seen. Inserting a rotor in the stator with the magnetic poles aligned 90 degrees anticlockwise to the induced stator magnetic poles makes for a correctly turning motor assembly.

Increasing the number of stator poles in a brushless DC (BLDC) can result in the following merits:

1. Higher Torque Density [6]: More stator poles create a stronger, more consistent magnetic field, resulting in greater torque output for the same motor size. This allows for a more compact design with higher power density.

2. Smoother Torque Production [6]: A higher number of stator poles leads to a more continuous and stable torque output, reducing torque ripple and vibration, which improves the motor's overall performance and efficiency.

3. Less Cogging Torque: Increasing the stator poles helps minimize cogging torque, leading to smoother operation, particularly at low speeds.

4. Better Efficiency [6]: More poles can reduce copper and iron losses and improve magnetic flux usage, enhancing the motor's overall efficiency.

At some point, increasing the number of poles is not favorable, since the back electromotive force (EMF) increases with an increasing number of motor poles. In equation 2.1, Faraday's law of induction is given, which states that a changing magnetic field creates an electric field along a contour X from a to b ($X = [a, b]$). Because of adding more motor poles, the same change of magnetic field, when turning the motor, now happens in less time, thus resulting in more back EMF and a reduced mechanical speed. Additionally, when the same mechanical output speed is required, while more motor poles are added, the electrical speed is higher. In this case a controller is needed that can produce these higher frequency signals, which can also become challenging when these frequencies become too high for standard micro controllers.

$$U_{ab} = \oint_a^b \vec{E} \cdot \vec{dl} = \frac{-d}{dt} \int_S \vec{B} \cdot \vec{dS} \tag{2.1}$$

The optimal number of stator poles is chosen based on specific application needs, balancing factors such as torque, speed, power density, cost, and size.

2.2 Motor Control Schemes

To achieve SHE, first the core part of a motor control scheme needs to be present, thus an understanding of a 'basic' motor controller design for a PMSM is needed, to which SHE techniques can be added. Two conventional controller designs are Field-Oriented Control

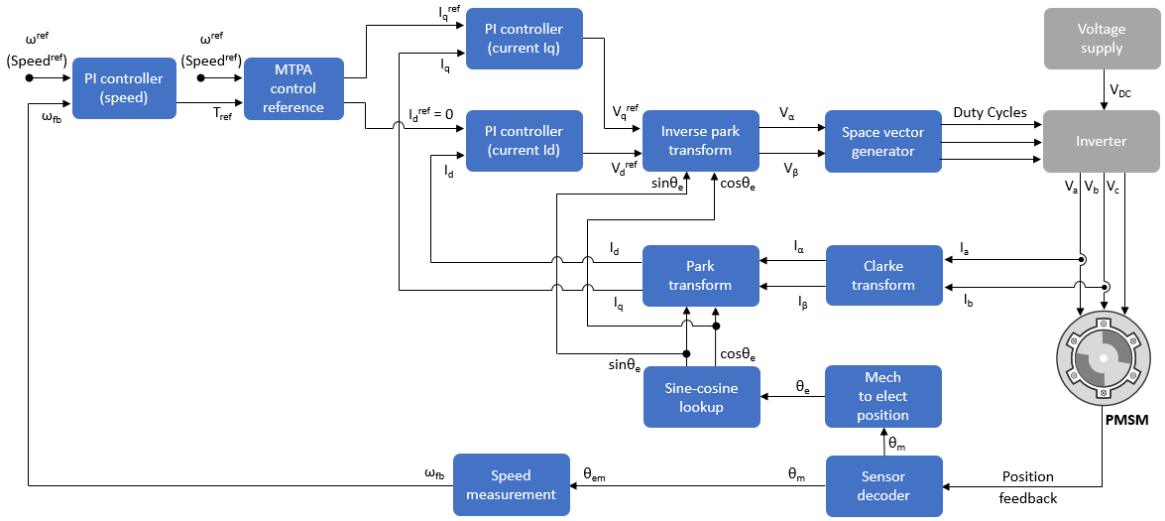


Figure 2.3: The FOC control scheme [7]

(FOC), pictured in Fig. 2.3 and Direct Torque Control (DTC), pictured in Fig. 2.5. The first uses (inverse) Park and Clarke transformations to convert rotating two phase system setpoints (a direct (d) and a quadrature (q) voltage setpoint; V_d and V_q) into a stationary reference frame (containing V_α and V_β). Lastly, the two phase system is converted to a three phase system (containing V_a, V_b and V_c). Gate signals can then be calculated using Sinusoidal Pulse Width Modulation (SPWM), a common technique to achieve correct gate

signal generation. SPWM works by comparing modulating signals to a certain carrier wave. The modulating waveforms are made by performing the Clarke transformation on V_α and V_β. The output is equal to the desired three phase sinusoidal voltages. Next, the three phase voltages are normalised to the supply voltage and compared to a triangle wave to obtain the gating signals for the inverter side of the motor controller. This SPWM algorithm can be seen for a given instance of time in Fig. 2.4 for further clarification.

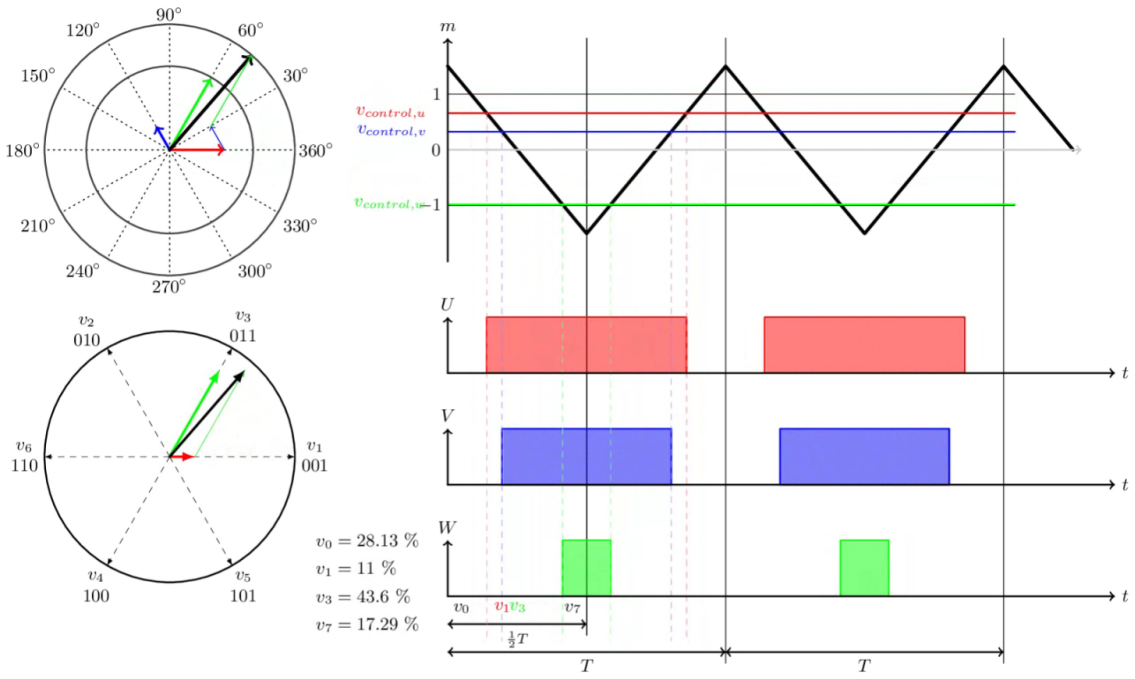


Figure 2.4: SPWM modulation vectors [8]

For the purpose of feedback, the three phase currents are measured (I_a , I_b and I_c) and multiplied by the inverse Clarke and inverse Park transformations. A stationary two phase system containing I_d and I_q is the result. Applying compensation to reach a certain reference setpoint is then done, usually with two Proportional Integral (PI) controllers to control the direct (d) and q axis current independently. An important remark is that inaccurate motor angle determination can make for noisy d,q variables or non-ideal operation. The Maximum Torque Per Ampère (MTPA) block present in Fig. 2.3 calculates the d and q axis current. In this report, a non-salient rotor is used, and thus the only function that the MTPA block has, is to limit currents to allowed values and to calculate a non-zero I_d reference, when the required operating speed is greater than the base speed of the machine. This control scheme thus incorporates field weakening already, where the motor synchronous frequency is increased. That is achieved by effectively advancing the motor timing, which limits the back EMF and allows for more q torque (and d torque) to be generated.

The required Park and Clarke transformations can be calculated by vector deconstruction and are given in equation 2.4 and 2.2. The inverse Park and Clarke transformations are stated in equations 2.5 and 2.3.

$$T_{\text{Clarke}} = \frac{2}{3}\begin{pmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{pmatrix} \tag{2.2}$$

$$T_{\text{Clarke}}^{-1} = \begin{pmatrix} 1 & 0 & 1 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & 1 \end{pmatrix} \tag{2.3}$$

$$T_{\text{Park}} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

$$T_{\text{Park}}^{-1} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

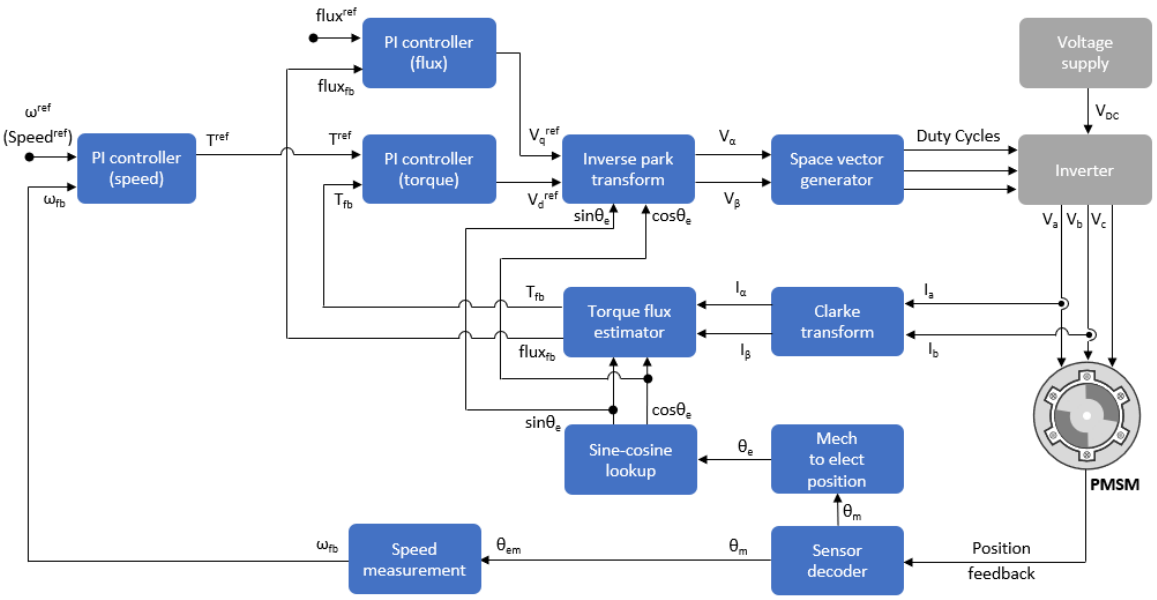


Figure 2.5: The DTC control scheme [9]

DTC directly controls the torque and flux of a motor without relying on decoupled control loops, which allows it to respond faster to changes in torque demand. The control is based on real-time estimations of torque and flux from measured three phase currents, passed through the Clarke transformation, bypassing the delays associated with modulation and PWM. FOC uses decoupled control loops (with PI controllers) to control the d-axis and q-axis currents. These currents control torque and flux, and the slower dynamic response is a result of the reliance on modulator and PWM. Thus, DTC eliminates the need for a modulator, which reduces the processing time and results in faster torque response [10]. DTC does not require position or speed sensors because it estimates the motor's flux and torque directly from voltage and current measurements. The need for an electrical motor

angle θ for the Park transformation is avoided because DTC doesn't rely on transforming current vectors into the stationary or rotating reference frame, unlike FOC. Instead, DTC operates in the stator flux frame and uses lookup tables to directly select voltage vectors based on the estimated torque and flux. It selects these different vectors when the hysteresis comparators trigger.

While PI controllers are present in some DTC implementations (e.g., for flux or torque regulation), their role is less critical compared to FOC. In DTC, the control is more direct and less dependent on the decoupling and linearization performed by PI controllers in FOC. In summary, DTC's dynamic response is faster because there is no need for a modulator, and it doesn't require position measurements because it operates based on real-time estimations rather than reference frame transformations.

Between the two options of FOC and DTC, FOC is a better starting point for the addition of SHE, because of several reasons. Overall, DTC has worse performance in terms of harmonics, compared to FOC, because of the variable switching frequency due to the previously mentioned hysterisis comparators. [10] Additionally, acoustic noise at low speeds and bad performance at low speeds are bad characteristics of DTC compared to FOC [10]. Advantages of DTC are that no position or speed measurements are needed, as the flux and torque are controlled directly from the output of the Clarke transformation (I_α and I_β) with the inputs being the measured three phase currents (I_a , I_b and I_c).

2.3 Selective Harmonic Elimination (SHE) Technique

The Selective Harmonic Elimination (SHE) techniques aim to improve the quality of the output voltage in pulse-width modulation (PWM) systems by minimizing specific harmonic components. When the harmonics of a signal are reduced, the Total Harmonic Distortion (THD), as defined in equation 2.6 improves. The improvement of THD has a positive effect on the effiency of the motor drive, as peak currents are reduced [11], [12]. For grid-tied inverters a power factor close to one may be desired. A lower THD helps in that case and a power factor closer to one can even reduce electricity bills [11].

$$THD = \frac{\sqrt{\sum_{n=2}^{\infty} V_{n\_rms}^2}}{V_{fund\_rms}} \tag{2.6}$$

The following description explains the process of implementing the first of the two used SHE techniques, from computing the Fourier series to adjusting the PWM-like signals to achieve harmonic elimination. The procedure that will be explained is similar to and inspired from [13].

2.3.1 SHE accomplished by Fourier computation and the equal area modulator

Fourier Series Computation

The process begins with computing the Fourier series of the phase voltages over a complete period. The Fourier series expansion of a periodic function $f(t)$ with period T is given by:

$$f(t) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{2\pi nt}{T}\right) + b_n \sin\left(\frac{2\pi nt}{T}\right)\right) \tag{2.7}$$

Given the symmetry and nature of the phase to neutral voltages of every phase, only the odd harmonics (i.e., sine components) are significant. Hence, the Fourier coefficients a_n (cosine terms) are zero, and the focus is on b_n. b_n is evaluated according to equation 2.8, where $V(t)$ represents the phase to neutral voltage of a singular phase.

$$b_n = \frac{2}{T} \int_0^T V(t) \sin\left(\frac{2\pi nt}{T}\right) dt \tag{2.8}$$

For the square wave phase to neutral voltages, the fourier series is derived in equation 2.9 and 2.10 , where the voltage is at zero volts from x_0 to x_1 and at the DC bus voltage from x_1 to x_2 alternatingly in the set of switching angles or timestamps $x_0....x_N$.

$$b_n = \frac{2}{T} \int_{x_0}^{x_1} 0 \sin\left(\frac{2\pi nt}{T}\right) dt + \frac{2}{T} \int_{x_1}^{x_2} V_{DC} \sin\left(\frac{2\pi nt}{T}\right) dt \tag{2.9}$$

$$b_n = \frac{2}{T} \left[-\frac{T}{2\pi n} V_{DC} \cos\left(\frac{2\pi nt}{T}\right)\right]_{x_1}^{x_2} = \frac{V_{DC}}{\pi n} (\cos(2\pi \frac{n}{T} x_1) - \cos(2\pi \frac{n}{T} x_2)) \tag{2.10}$$

These b_n coefficients represent the harmonic content of the voltage signal, with each b_n corresponding to the amplitude of the n-th harmonic. Because the phase to neutral voltages exhibit quarter wave symmetry, the even harmonic components are zero and thus equation 2.10 is only relevant for odd harmonic components. The Fourier series can also be computed by taking into account quarter wave symmetry properties of the modulated waveform. The start of the derivation is given in equation 2.11

$$b_n = \frac{8}{T} \int_0^{T/4} V(t) \sin\left(\frac{2\pi nt}{T}\right) dt \tag{2.11}$$

$$b_n = \frac{8}{T} \sum_{i=0}^{i=I} (\int_{x_i}^{x_{i+1}} 0 \sin\left(\frac{2\pi nt}{T}\right) dt + \int_{x_{i+1}}^{x_{i+2}} V_{DC} \sin\left(\frac{2\pi nt}{T}\right) dt) \tag{2.12}$$

$$b_n = \frac{2}{T} [-\frac{T}{2\pi n} V_{DC} \cos\left(\frac{2\pi nt}{T}\right)]_{x_1}^{x_2} = \frac{V_{DC}}{\pi n} (\cos(2\pi \frac{n}{T} x_1) - \cos(2\pi \frac{n}{T} x_2)) \text{ for } n = 1 + 2 * i, i \in N \tag{2.13}$$

Various SHE methods

Implementing Selective Harmonic Elimination requires solving a non-linear problem set. A conventional linear set of equations can be solved by performing several operations and eliminating variables, which can then be used to substitute and solve the set of equations. As long as there are as many equations as individual variables, the equations are consistent and all equations are independent, the system is solvable. An example of an equation set that could be solved to achieve SHE is given in equation 2.14. The system of equations includes four harmonics that should be eliminated and it can be seen that this system is already complex to solve algebraically. A more realistic set of like twenty switching angles per period, would result in every equation of the set having at least five terms (because of quarter wave symmetry). Solving such a system of equations and especially when more harmonics need to be eliminated, is not trivial as this requires long derivations. Thus solving such a problem set is usually done by using numeric methods, in which a first guess is made for a solution set and iteratively a solution to the problem is found.

$$\begin{cases} H_{c5} = \dfrac{V_{DC}}{\pi n}(\cos(2\pi\dfrac{5}{T}x_1) - \cos(2\pi\dfrac{5}{T}x_2) + \cos(2\pi\dfrac{5}{T}x_3) - \cos(2\pi\dfrac{5}{T}x_4) + ...) \\ H_{c7} = \dfrac{V_{DC}}{\pi n}(\cos(2\pi\dfrac{7}{T}x_1) - \cos(2\pi\dfrac{7}{T}x_2) + \cos(2\pi\dfrac{7}{T}x_3) - \cos(2\pi\dfrac{7}{T}x_4) + ...) \\ H_{c11} = \dfrac{V_{DC}}{\pi n}(\cos(2\pi\dfrac{11}{T}x_1) - \cos(2\pi\dfrac{11}{T}x_2) + \cos(2\pi\dfrac{11}{T}x_3) - \cos(2\pi\dfrac{11}{T}x_4) + ...) \\ H_{c13} = \dfrac{V_{DC}}{\pi n}(\cos(2\pi\dfrac{13}{T}x_1) - \cos(2\pi\dfrac{13}{T}x_2) + \cos(2\pi\dfrac{13}{T}x_3) - \cos(2\pi\dfrac{13}{T}x_4) + ...) \end{cases} \tag{2.14}$$

The equal area modulator

The equal area modulator is used in this SHE technique. This modulator can be best explained by referring to Fig. 2.6, where the principle of the equal are criteria is displayed. During one off and one on state of the output waveform, the area below the target sinusoidal signal, must be equal to the area above the target signal, since the resulting square wave during this time frame, will best match the target signal. To perform the complete modulation, first a set of σ_k's need to be chosen. In this implementation they are divided evenly to start with and corresponding θ_k's are calculated afterwards. To calculate these θ_k's, a derivation is given in equations 2.15 to 2.17.

$$\int_{\sigma_{k-1}}^{\theta_k} V_{DC} - (V_{DC}/2 + A\sin(x))dx = \int_{\theta_k}^{\sigma_k} V_{DC}/2 + A\sin(x)dx \tag{2.15}$$

$$[V_{DC}/2 \cdot x + A\cos(x)]_{\sigma_{k-1}}^{\theta_k} = [V_{DC}/2 \cdot x - A\cos(x)]_{\theta_k}^{\sigma_k} \tag{2.16}$$

$$\theta_k = \frac{\sigma_k}{2} + \frac{\sigma_{k-1}}{2} + \frac{A(\cos(\sigma_{k-1}) - \cos(\sigma_k))}{V_{DC}} \tag{2.17}$$

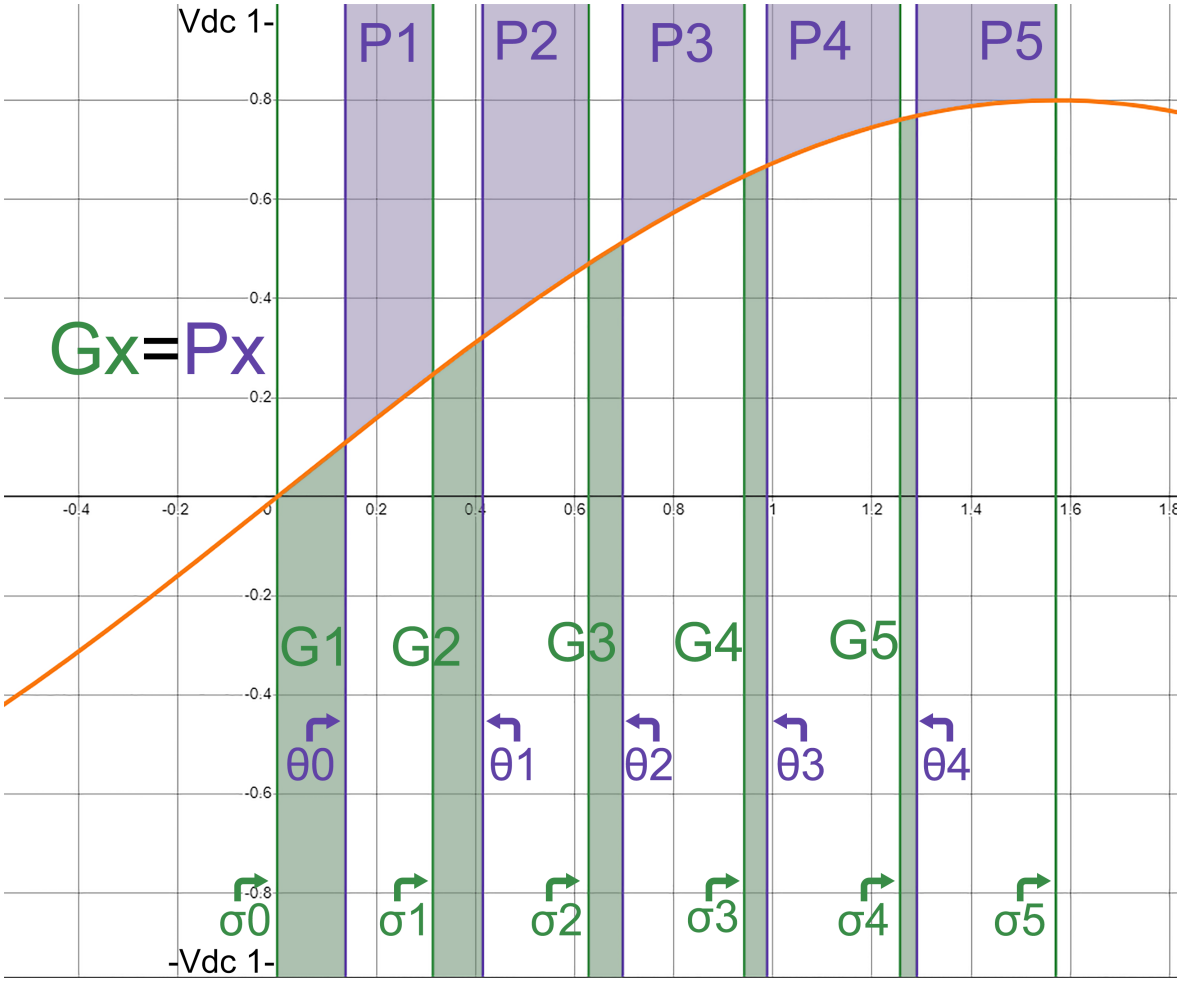


Figure 2.6: The equal area criteria is pictured, which is used in the equal area modulator. The area above the modulating waveform (purple) should be equal to that under the modulating waveform (green). Additionally, the σ_k's and θ_k's are depicted in the figure as well.

Harmonic Compensation Using Switching Angles

Once the harmonic coefficients are determined, the next step is to adjust the switching angles to achieve harmonic elimination. The goal is to modify the switching instants so that the undesired harmonics are minimized or nullified.

The switching instants (angles) are denoted as $\theta_1, \theta_2, \ldots, \theta_k$ within one quarter of the period T. Due to quarter-wave symmetry and half-wave symmetry of the waveform, the complete set of switching angles for the entire period can be derived from these primary angles.

For SHE, we solve a system of nonlinear equations, which are based on the Fourier series coefficients. These equations relate the switching angles to the harmonic content:

$$\sum_{k=1}^{K} \cos(n\theta_k) = \frac{4V_{dc}}{n\pi} \quad \text{for} \quad n = 1, 3, 5, \ldots, N \tag{2.18}$$

Where: - K is the number of switching angles per quarter period. - V_{dc} is the DC link voltage. - n are the harmonic orders to be eliminated (odd harmonics).

These equations must be solved simultaneously to find the optimal switching angles θ_k that minimize or eliminate the specified harmonics.

Iterative Solution of Nonlinear Equations

The system of nonlinear equations is typically solved using numerical methods such as the Newton-Raphson method or other iterative techniques. The iterative process starts with an initial guess for the switching angles and refines these guesses until the equations are satisfied within a specified tolerance.

$$\theta_k = \theta_{k,\text{previous iteration}} + \sum_{3}^{5,7,11\ldots} \frac{\frac{h_m}{m}(\cos(\sigma_k) - \cos(\sigma_{k-1}))}{V_{DC}} \tag{2.19}$$

1. **Initial Guess:** Begin with an initial guess for the switching angles, like is mentioned in section 2.3.1.
2. **Iterative Compensation:** Calculate the Fourier harmonic constants of the harmonics that need to be eliminated. Subtract these harmonics from the target signal, which is then used in the equal area modulator to compute a new set of switching angles. The equal area modulator then includes the term from equation 2.19 for every harmonic that needs to be eliminated and on every iteration.
3. **Convergence Check:** Continue the iterations until the changes in the switching angles between successive iterations are below a predefined threshold.

Gating Signal Generation

After determining the optimal switching angles, the gating signals are generated accordingly. The switching times corresponding to these angles are used to control the power electronic switches, ensuring that the output voltage waveform meets the desired harmonic profile.

Conclusion

The implementation of the SHE technique involves computing the Fourier series of the phase voltages, determining the coefficients for specific harmonics, and adjusting the switching angles to minimize or eliminate those harmonics. This process requires solving a system of nonlinear equations iteratively to find the optimal switching angles. The resulting signals ensure improved output voltage quality by selectively eliminating undesired harmonics.

2.3.2 Accomplishing SHE using the Least Mean Square (LMS) method

In [14], the authors present a cutting-edge approach to mitigating harmonics in power electronic systems. This approach is built around the development of an Adaptive Selective

Harmonic Elimination (ASHE) algorithm, which is applied in a systematic and non-intrusive manner to remove specific undesirable harmonic components from control variables such as current or voltage. The key innovation here is that the harmonic filtering is achieved without compromising the dynamic response of the system, which remains under the control of conventional controllers, like proportional-integral (PI) regulators.

Harmonic disturbances and the intertwining with the rest of the control loop

In power electronics, harmonic distortions arise due to various sources, such as blanking time in Pulse Width Modulation (PWM), utility voltage distortion, or non-linearities within the system. These distortions often manifest as higher-frequency harmonic components, which regulators like PI controllers cannot easily reject. This is particularly true when the regulators operate in the discrete domain, where the limitations in bandwidth due to sampling rates become significant.

Concept of the algorithm

THe method builds on existing harmonic elimination techniques but introduces a novel aspect: the ASHE algorithm. This algorithm adapts methods from digital signal processing (DSP), where single-frequency components are eliminated via adaptive filters, and applies them to power electronics control. The fundamental aspect of ASHE is its ability to cancel specific harmonic frequencies without significantly affecting the rest of the system's behavior, which makes it ideal for a range of applications such as motor drives or uninterruptible power supplies.

In the ASHE algorithm, the central concept involves creating a reference signal at the harmonic frequency that needs to be eliminated. This reference signal is processed using adaptive weights and then subtracted from the original signal to eliminate the unwanted harmonic. The algorithm adapts over time to match the phase and amplitude of the harmonic component, ensuring it is effectively canceled. Notably, the ASHE algorithm can operate in both synchronous and stationary reference frames, preserving the adaptability of conventional synchronous reference frame regulators. [14]

The key ingredient is the LMS algorithm

This ASHE algorithm uses the Least Mean Square (LMS) algorithm in its architecture. An LMS algorithm minimizes an error signal, which in the case of SHE represents the harmonic component to be eliminated. The LMS operates by continuously adjusting the weights associated with the reference signal's sine and cosine components. These adaptive weights ensure that the reference signal is dynamically aligned with the harmonic disturbance, which means that this disturbance can be eliminated.

A key strength of the LMS-based approach is its simplicity and effectiveness. The algorithm updates the weights based on a gradient descent method, allowing it to converge relatively quickly and with minimal computational effort. In the context of power electronics,

this means that the ASHE algorithm can be implemented without significantly increasing the complexity of the control system.

An LMS algorithm can be implemented in various ways. For example double integration of the error signal can be performed, however this technique incorporates single integration of the error signal resulting in the used equations, given in equation 2.20. In this equation, a new weight component is calculated by multiplying the error signal (ϵ_k) by a gain constant (2μ) and the sinusoidal signal (x_k) and integrating that by adding it to the old weight constant (w_k). The sinusoidal signal represents the sine or cosine component. The combining of these two weighted sinusoidal signals effectively results in the output being a sinusoid with a matched phase to the harmonics present in the current waveforms. Obtaining the phase of the fundamental signal, presents the need for accurate phase detection. This can be done by implementing a phase locked loop, which itself can be made by similar techniques, like the LMS algorithm.

$$w_{k+1} = w_k + 2\mu\epsilon_k x_k \tag{2.20}$$

Integration of ASHE with the control loop

One of the critical challenges in applying selective harmonic elimination techniques is ensuring that they do not interfere with the primary control loop, which is responsible for the system's overall dynamics. This is addressed by assuming that the ASHE algorithm operates on a slower time scale than the primary control loop. This ensures that the harmonic elimination process does not introduce instability or degrade the performance of the system's main control objectives, such as fast transient response.

The ASHE filter is integrated with the plant's control system by injecting a signal that cancels the targeted harmonic components. Importantly, the plant's transfer function is taken into account, which ensures that the injected signal is properly shaped to cancel the harmonics in the plant's output. This is done by incorporating an inverse model of the plant's transfer function into the ASHE algorithm. For instance, if the plant is modeled as an RL (resistance-inductance) circuit, the ASHE filter adjusts its output to match the harmonic component's behavior in such a system, allowing for effective cancellation. [14]

Application in regenerative converters

To demonstrate the effectiveness of the ASHE method, the author of [14] applies the technique to a regenerative voltage source converter. This type of converter is often used in systems like motor drives and UPS, where harmonic elimination is crucial for maintaining power quality and system performance.

In the simulation example, the converter is subject to harmonic distortion due to dead time in the PWM control signals. Without compensation, this dead time introduces significant fifth and seventh harmonics into the current waveform. By applying the ASHE algorithm, these harmonics are successfully eliminated, which makes for a much cleaner current signal

with minimal distortion.

Interestingly, the ASHE algorithm was able to achieve this without modifying the fundamental control loop of the converter. This highlights the advantage of the method: it operates as an add-on to the existing control system, requiring minimal changes to the primary control logic. The effectiveness of the ASHE algorithm in eliminating specific harmonics was confirmed by observing the current spectrum before and after applying the filter, with significant reductions in the fifth and seventh harmonic components.

Elimination of additional harmonics

The ASHE algorithm is not limited to the elimination of just a single harmonic. The methodology can be extended to eliminate multiple harmonic components simultaneously. This is achieved by creating multiple ASHE blocks, each tuned to and targeting a specific harmonic frequency. The use of multiple blocks allows for the simultaneous cancellation of several harmonics, making the system adaptable to a wide range of unwanted harmonics.

Conclusion and effects

The ASHE algorithm presents itself as a viable option in the field of harmonic elimination for power electronics. Its ability to selectively cancel harmonic components without interfering with the primary control loop makes it an attractive option for a wide range of applications. By leveraging adaptive filtering techniques from digital signal processing, the ASHE algorithm offers a robust and efficient solution to the challenge of harmonic distortion in power systems.

For practitioners seeking to implement harmonic elimination in systems such as motor drives, UPS, or regenerative converters, the ASHE algorithm provides a flexible and effective tool. Its ease of integration, combined with its effectiveness in eliminating multiple harmonics, makes it a valuable addition to conventional control systems. Furthermore, the fact that it does not require detailed knowledge of the plant's parameters enhances its practicality in real-world applications where system characteristics may vary over time.

The next steps in implementing this technique would involve tuning the ASHE algorithm for specific systems, adjusting the adaptation gain for optimal convergence, and potentially extending the method to address additional harmonics as required. With proper implementation, the ASHE algorithm could significantly improve the performance of power electronic systems by reducing harmonic distortion and enhancing overall power quality.

2.4 Third Harmonic Injection

Third harmonic injection (THI) is a commonly used technique to gain performance in three phase star connected motor drives with a floating ground point. The technique is compatible with various modulation schemes, like Space Vector Pulse Width Modulation (SVPWM),

where modulating signals are compared to a carrier wave to generate appropriate gate signals. Third harmonic injection introduces an additional component to the modulating signal. The aim is to enhance the utilization of the DC bus voltage, which leads to improved performance in a PMSM.

In conventional PWM modulation schemes, the reference signal is purely sinusoidal. However, this approach underutilizes the available DC bus voltage because the peak of the sinusoidal wave is less than the peak value that could be reached if the waveform utilized the full DC bus range. By injecting a third harmonic component, the peak of the waveform is increased without altering the fundamental frequency, thus maximizing the utilization of the DC link voltage. The waveforms resulting from using the THI technique can be seen in Fig. 2.7. To implement third harmonic injection in a motor control system, the modulating

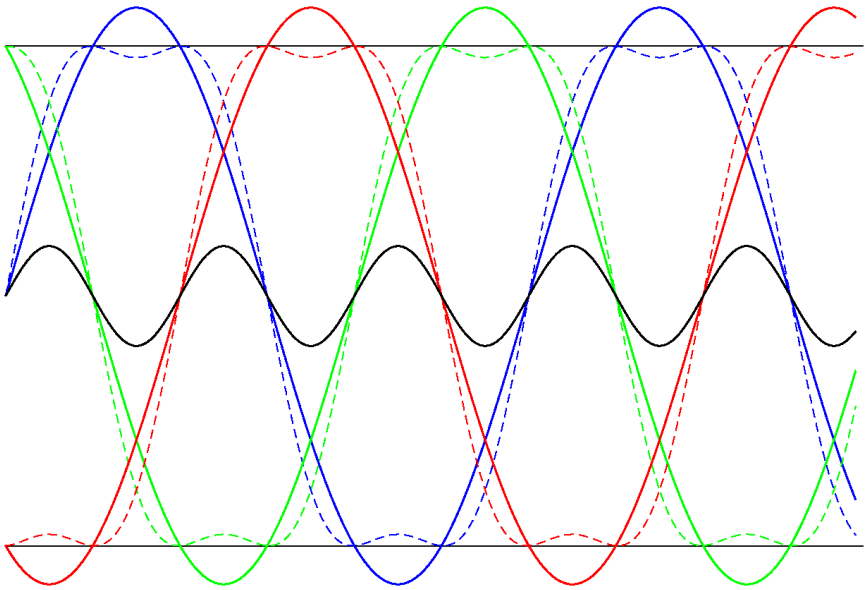


Figure 2.7: The image presents an illustration of the three phase output voltages of an inverter with THI, seen as the dotted lines. The waveforms shown with the solid lines are the phase to neutral voltages of the three phase system. The black waveform shows the third harmonic that is injected, where this signal is common to all three phases, which is exactly why THI works. [[8]]

signals for each phase are altered by adding the calculated third harmonic component. The modified signals are then fed into the PWM generator, which produces the gate signals for the inverter. This process enhances the voltage output without requiring a higher DC bus voltage, which is particularly beneficial in applications where the supply voltage is fixed or where efficiency is paramount.

By integrating third harmonic injection into a motor control scheme, the voltage margin is effectively increased, which allows for a higher maximum phase to neutral voltage from the inverter. This leads to improved torque and speed capabilities of the motor, especially in applications where maximizing efficiency and performance is critical. Moreover, the injection of the third harmonic does not require additional hardware changes, making it a cost-effective

enhancement to conventional PWM techniques.

The introduction of third harmonic injection can be seamlessly integrated into existing control schemes such as Field-Oriented Control (FOC) or Direct Torque Control (DTC). In these systems, the third harmonic is added after the Park and Clarke transformations. This modified signal is then normalized and compared to the carrier wave in the SPWM algorithm, as depicted in Fig. 2.4.

When viewing Fig. 2.7 the question arises what the amplitude of the third harmonic should be to gain the best injection performance and bus utilisation. When the amplitude of the injected wave is smaller than optimal, the bus utilisation is not maximised and conversely, when the amplitude of the injected wave is larger than optimal the same situation plays out, as the 'side lobes' of the waveform increase in amplitude. A derivation for this amplitude is given in equations B.1 to B.1.

Derivation of the third harmonic to be injected

A start is made by writing down the equation for the modulating signal with the superpositioned third harmonic.

$$m(x) = sin(x) + asin(3x), \mathbb{D} : \{x | x \in [0, 2\pi]\} \tag{2.21}$$

The derivative is computed with respect to x.

$$m'(x) = cos(x) + 3acos(3x) \tag{2.22}$$

The minima and maxima are of interest:

$$m'(x) = 0 \tag{2.23}$$

To solve the equation

$$\cos(x) + 3a\cos(3x) = 0 \tag{2.24}$$

for x, the derivation is provided in section B.

2.5 Model Predictive Control schemes

In recent years, MPC has shown its potential as a proper approach for managing the complex control challenges presented by various topologies of motor controllers. This section provides a detailed dive into a couple of MPC methods and provides a comparison of the common MPC methods as well, addressing their strengths and weaknesses across multiple metrics. The metrics evaluated in this comparison include computational burden, design complexity, ability to accommodate multiple control objectives, tuning efforts for weighting factors (WFs) and controller gains, switching frequency, sensitivity to parameter mismatches, steady-state performance, and dynamic performance.

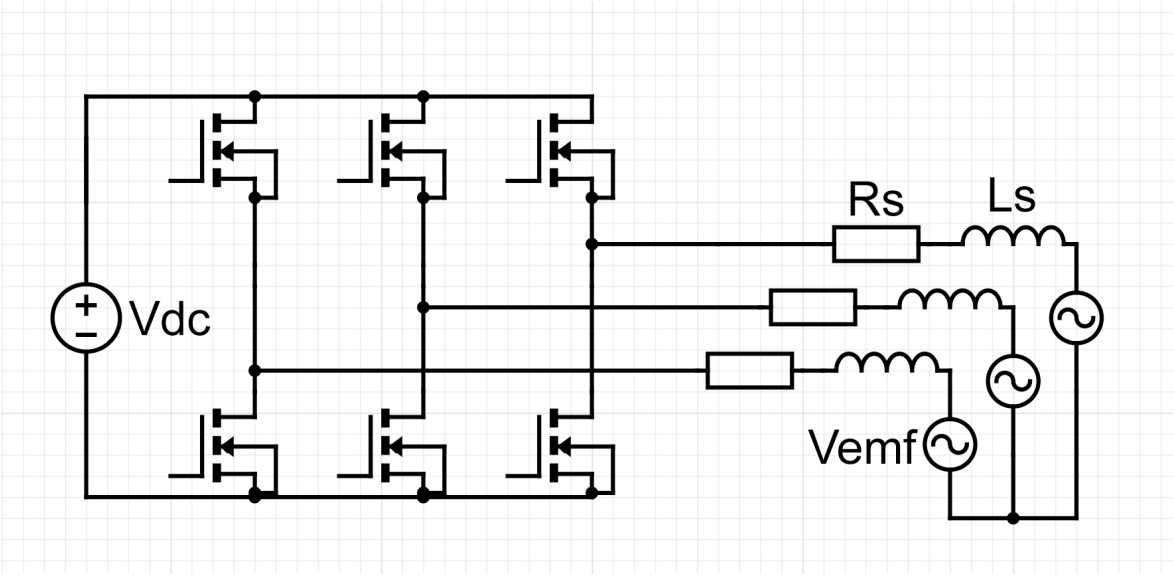


Figure 2.8: Circuit diagram of the system that is used

2.5.1 System model for a PMSM machine

MPC schemes rely on a model of the motor system to make predictions on what voltage (vector) should be applied to the inverter that controls the motor. A schematic of the system to model is presented in Fig. 2.8. Analysing this schematic in the stationary α, β frame, and utilising Kirchhoffs law together with the state equation for inductor current, yields equation 2.25. The resulting model is similar to the one presented in [15].

$$\begin{aligned} L\frac{di_\alpha}{dt} &= U_\alpha - Ri_\alpha - U_{\alpha,emf} \\ L\frac{di_\beta}{dt} &= U_\beta - Ri_\beta - U_{\beta,emf} \end{aligned} \tag{2.25}$$

Equation 2.25 is integrated to reach the equations related to predicting the α and β currents. In an ideal case, a MPC only requires one horizon to make an accurate current prediction. However, since microcontrollers or computers take some time to finish a calculation, a second horizon needs to be predicted as well, so that the calculation time is accounted for as well. The equations for the predicted currents are available in equations 2.26 and 2.27 [15], [16].

$$\begin{aligned} i_{p,\alpha}(k+1) &= \left(1 - \frac{RT_s}{L}\right) i_\alpha(k) + \frac{T_s}{L}\left[U_\alpha(k) - U_{\alpha,emf}(k)\right] \\ i_{p,\beta}(k+1) &= \left(1 - \frac{RT_s}{L}\right) i_\beta(k) + \frac{T_s}{L}\left[U_\beta(k) - U_{\beta,emf}(k)\right] \end{aligned} \tag{2.26}$$

$$\begin{aligned} i_{p,\alpha}(k+2) &= \left(1 - \frac{RT_s}{L}\right) i_\alpha(k+1) + \frac{T_s}{L}\left[U_\alpha(k+1) - U_{\alpha,emf}(k+1)\right] \\ i_{p,\beta}(k+2) &= \left(1 - \frac{RT_s}{L}\right) i_\beta(k+1) + \frac{T_s}{L}\left[U_\beta(k+1) - U_{\beta,emf}(k+1)\right] \end{aligned} \tag{2.27}$$

Depending on what type of MPC will be used, a cost function may or may not be needed to complete the control scheme. In case optimising a control function is necessary for that specific MPC type, one is stated in equation 2.28, where it is stated that for a minimal cost, the goal i_α must be equal to the predicted i_α [15].

$$C(k) = \|i^*_\alpha(k+2) - i_{p,\alpha}(k+2)\|^2 + \|i^*_\beta(k+2) - i_{p,\beta}(k+2)\|^2 \tag{2.28}$$

2.5.2 Conventional FCS-MPC (Finite Control Set MPC)

- Advantages: Simplicity in the design of the controller scheme. Allows for the incorporation of multiple control targets.
- Disadvantages: Requires a distinct WF for each objective, leading to significant tuning efforts. The variable switching frequency complicates filter design, making it less suitable for grid-connected applications.

Principles of FCS-MPC

FCS-MPC operates by determining the optimal switching sequence over a predefined control period (T_s), based on the current system state and a cost function that typically includes terms for compensation of measured parameters and control variables. The absence of a modulator in FCS-MPC allows for immediate and precise control actions. However, this approach results in only one switching vector being applied for the entire control interval, that means increased voltage and current ripples will result. The high ripples in the output waveform can significantly degrade steady-state performance compared to PWM-based methods, which utilize varying duty cycles to obtain a smoother output. [16], [17]

Harmonic Spectrum and Filter Design Challenges

One effect of the variable switching frequency associated with FCS-MPC is the wide harmonic spectrum generated in the output current and voltage. This broad spectrum can complicate filter design, as filters must attenuate not only a specific target frequency, but also a range of harmonics that can interfere with the system's feedback and cause equipment malfunctions. Further negative consequences are that increased ripples lead to higher levels of harmonic distortion, affecting the overall quality of the power delivered.

Solutions to Address Variable Switching Frequency

The Hybrid FCS-MPC approach is one of the earliest solutions proposed to mitigate the issues associated with variable switching frequency. This method involves the application of a low-pass filter as a demodulation stage after the FCS-MPC controller. By filtering out high-frequency components, the hybrid technique reduces the harmonic distortion at the output. Subsequently, a sinusoidal PWM or space vector modulation stage can be employed to smooth the output further. This method is practical and intuitive, but it does result in

some degradation of dynamic and steady-state performance compared to traditional FCS-MPC. Various MPC methods that are to be explained will also address the issues of variable switching frequencies, namely M2PC and OSS-MPC [1].

Control Steps of FCS-MPC

The FCS-MPC operates according to a few specific steps:

1. Constructing a discrete-time prediction model
2. Designing the cost Function
3. Applying the optimal vector through an optimization algorithm

For this MPC method, a discrete time prediction system model like laid out in section 2.5.1 can be used to evaluate the possible switching vectors.

Integrating a cost function g is a key component in FCS-MPC, defining the optimization problem by encapsulating control objectives and system constraints. The formulation of the cost function is crucial, especially in systems with multiple objectives, as it directly influences system performance and stability. An example for a cost function that includes multiple objectives or targets ($i^* = i_p$, $v_x^* = v_x$, $v_x^* = v_x$ and a minimum deviation in v_z) is given in equation 2.29. Every λ_i is a weighing factor that specifies which objectives are the most important or should weigh heavier in the cost function [17].

$$g = [i^*(k+1) - i_p(k+1)]^2 + \lambda_1[v_x^*(k+1) - v_x(k+1)]^2 + \lambda_2[v_y^*(k+1) - v_y(k+1)]^2 + \lambda_3[\Delta v_z(k+1)]^2 \tag{2.29}$$

The objectives can be either a topology-related one or a application-related one, where the first are integrated to ensure proper inverter operation, such as dc-link voltage balancing and capacitor voltage control and the latter type of targets are used to be compatible with the specific requirements of the load, including reference current tracking, torque control, and power management [17].

The weighting factors (λ) play a significant role in tuning the control behavior, and their proper selection is critical for system stability and performance.

2.5.3 Optimal Vector Identification through Optimization Algorithm

After setting up the discrete-time model and cost function, an optimization algorithm identifies the optimal control action. The traditional approach often employs an Exhaustive Search Algorithm (ESA), but in multilevel inverter applications, the number of states can lead to significant computational demands.

For a five level inverter, each phase has eight switching states, resulting in 512 (i.e., 8^3) states for the three-phase implementation. This necessitates extensive computations—512×512—to evaluate the control variables and the cost function for each control sampling period.

Due to this computational burden, alternative optimization algorithms are explored, including sphere decoding algorithms (SDA) and other techniques designed to alleviate the weight of the optimization problem, particularly for longer prediction horizons. [1]

2.5.4 Long-Horizon MPC

- Advantages: Superior steady-state performance due to a more comprehensive prediction horizon.
- Disadvantages: Exponential increase in computational burden with the increase in prediction steps

In scenarios requiring long-term predictions of system behavior, Long-Horizon FCS-MPC is an innovative solution. By manipulating the switch positions over an extended prediction horizon, the optimization problem can become complex. However, recent studies demonstrate that extending the prediction horizon can significantly enhance control performance, particularly in complex MLI applications. [1]

Benefits of Long Prediction Horizons

1. Improved Closed-Loop Performance: Utilizing a longer prediction horizon allows for more accurate modeling of system dynamics, enabling the controller to make more accurate decisions about future states.
2. Reduced Total Demand Distortion (TDD): Longer horizons have been shown to decrease current Total Demand Distortion (TDD), thereby aligning with grid standards and enhancing system efficiency.
3. Adaptability to Complex Dynamics: The ability to predict the evolution of the system state over a longer time interval is particularly beneficial for higher-order systems, such as those utilizing Modular Multilevel Converters (MMCs) or systems with coupled inductors. [1]

Computational Challenges and Solutions

The benefits of long-horizon FCS-MPC are clear, but the computational complexity associated with evaluating a broad set of potential switching sequences poses a significant challenge for real-time implementation. To address this issue, several methods have been proposed:

- Sphere Decoding Algorithm (SDA): This approach reformulates the optimization problem into an Integer Least-Squares (ILS) problem, reducing computational load while maintaining an optimal solution. The SDA narrows down the search space by looking at probable candidates to find the optimal switching sequence.

- Move Blocking Strategies: By grouping multiple prediction moves, these strategies reduce the effective prediction horizon, which balances performance with computational efficiency.
- Heuristic Approaches: Simplifying the optimization process through heuristics can speed up the process without drastically compromising control quality. [1]

2.5.5 Lyapunov-based MPC

- Advantages: Uses a discrete form of the derivative of a positive Lyapunov cost function to ensure system stability and robustness. Shares similarities with conventional FCS-MPC but with optimized weights.
- Disadvantages: Not suitable for systems with multiple constraints, lacking detailed stability analysis. [1]

Lyapunov-based MPC is a model predictive control strategy that uses a discrete form of the Lyapunov function to ensure system stability. The Lyapunov function is a mathematical tool that helps verify the stability of a system by measuring how system states evolve over time. In Lyapunov MPC, the control outputs are designed to decrease the value of this function at each step, ensuring that the system remains stable. [1]

2.5.6 M2PC (Modified Model Predictive Control) and OSS-MPC (Optimal Switching Strategy MPC)

- Advantages: Enhanced steady-state performance. Fixed switching frequency reduces the complexity associated with filter design.
- Disadvantages: Increased computational load and complexity compared to traditional methods.

Another promising method is the Modulated MPC (M2PC), which simulates PWM-like behavior by allowing multiple vectors to be applied within a single control cycle. The durations for which each vector is applied are inversely proportional to their associated cost function values, optimizing performance in terms of harmonic distortion and dynamic response. This method has shown efficacy across various MLI topologies, including Neutral Point Clamped (NPC) converters and Cascaded H-Bridge (CHB) converters, enhancing flexibility in control applications. [1]

Optimal Switching Sequence MPC (OSS-MPC) The Optimal Switching Sequence MPC (OSS-MPC) focuses on identifying the best sequence of switching states from a predetermined set. The converter states are categorized into a finite number of sequences, and the optimal sequence is determined by optimizing a cost function across these sequences. While OSS-MPC has shown improvements in steady-state performance and harmonic spectrum compared to standard FCS-MPC, it takes on a higher computational burden due to the increased complexity of identifying the optimal sequence. [1]

2.5.7 Indirect MPC (CCS-MPC)

- Advantages: Retains the multiple objectives handling of MPC while allowing for explicit system constraints. Has a modest computational complexity.
- Disadvantages: More WFs than FCS-MPC can complicate tuning. Susceptible to model deviations and parameter uncertainties.

Where regular FCS-MPC takes on the task of the modulation as well, Indirect MPC specifically does not. The modulation can be performed seperately, which means that a regular SPWM block could be used to convert the modulating signals to gating signals. [1]

2.5.8 DB-MPC (Discrete-Time Base MPC) or (Dead-Beat MPC)

- Advantages: Achieves high dynamic performance and steady-state characteristics similar to PI controllers. Maintains a constant switching frequency.
- Disadvantages: Difficulty in effectively addressing multiple objectives without complexity.

This type of MPC is a subgroup of Indirect MPC. A dead-beat controller has the property of reaching a control target in one sampling period, thus only one sampling period is off target i.e. a dead beat. [1], [18]

2.5.9 Future Trends in MPC for MLIs [1]

Despite the advancements in MPC methods for inverters, several challenges remain that are critical for future research. Below are key research directions that could enhance the performance and usefulness of MPC in practical scenarios:

Computational Efficiency

The computational burden of MPC, especially for OSS-MPC, poses challenges for real-time applications. Future research should focus on reducing the computational demands without degrading performance, particularly for high-level MLIs (e.g., $N_{level} >= 5$).

Formulation of Nonlinear MPC

Many MPC formulations currently incorporate linear systems. Moving towards a framework that addresses the nonlinear nature of many applications can lead to more robust control strategies.

Tuning of Weighting Factors (WFs)

The process of tuning WFs is time-consuming. Integrating Artificial Neural Networks (ANNs) for optimizing WFs could facilitate real-time adjustments, making FCS-MPC more adaptive without adding computational complexity.

Reliability and Component Stress Distribution

Investigating the relationship between component performance and control strategies could lead to improved reliability and longer lifetimes of semiconductor devices and capacitors used in MLIs.

Chapter 3

Implementation

A starting point was obtained by building upon an existing demo model available in the PLECS software [19]. In this model, a carrier based FOC controller is realised using PI controllers. Modifications can be made to this model to replace the PI controllers or the carrier based modulator.

The final code that was made to simulate the system can be viewed in section G.

3.1 MPC implementation

In indirect MPC, the control system generates modulating signals, which are fed into a modulation method (such as SPWM or SHE) to determine the switching states of the inverter. Unlike direct MPC, which directly computes the switching states, indirect MPC allows for more flexibility, particularly in the modulation strategy.

This implementation focuses on regulating the output voltages and currents of a two level inverter. By utilizing both the stationary $\alpha\beta$ frame and the rotating dq frame, the controller is compatible with the SHE method and the SPWM modulation. The model presented in section 2.5.1 is used to calculate and predict values for the $\alpha\beta$ currents.

3.1.1 Adjustable Prediction Horizon

One of the properties of this implementation is the ability to modify the prediction horizon. The prediction horizon in MPC defines how many future time steps the controller considers in its optimization process.

- **Short Horizon:** This provides faster computational times and a more reactive control response but may sacrifice long-term optimization.
- **Long Horizon:** A longer horizon improves long-term control quality but increases computational demand.

This adjustability provides flexibility, allowing the controller to adapt based on the application's computational capacity and dynamic requirements.

If such an MPC controller is implemented in hardware, calculation time is needed as a processor takes time to complete its computations. This calculation time presents itself as a delay in the PLECS simulation. One prediction horizon is thus not enough to get a correctly operating control loop. In the simulation a delay block is implemented in the modulator. This block has a delay of one sampling period, which is equal to the inverse of the PWM frequency specified for the modulator. The MPC controller itself runs at an update frequency of 100 kHz. So if the PWM frequency is set at 10 kHz, the MPC needs to predict $1 + \frac{1}{10}/\frac{1}{100} = 11$ horizons.

3.1.2 Hybrid Control in the $\alpha\beta$ and dq Frames

Control in the $\alpha\beta$ Frame

The stationary $\alpha\beta$ frame is used to compute and control the voltages in a two-phase plane, simplifying the system's mathematical representation. This frame avoids the trigonometric complexities inherent in transformations, making it useful for high-speed changes in reference signals.

Transformation to the Rotating dq Frame

To complement the $\alpha\beta$ frame, voltages are also transformed into the rotating dq reference frame using the Park transformation matrix, which was previously stated in equation 2.4:

$$\begin{pmatrix} V_d \\ V_q \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} V_\alpha \\ V_\beta \end{pmatrix} \tag{3.1}$$

where θ is the angular position of the rotating frame. As can be seen from equations 2.26 and 2.27, the variables that are inputs to the equation require to be altered in such a way that they represent themselves in a future timestamp. In the code this is implemented by performing an inverse Park transformation of the dq components, together with the motor angle θ, but a term of $\frac{1}{100000} * \omega_{synchronous}$ is added to the motor angle for every new horizon prediction. This ensures that the $U_{\alpha\beta,emf}$, $U_{\alpha\beta}$ and $I_{\alpha\beta}$ input variables are correctly evolved in time.

3.1.3 Integration with Selective Harmonic Elimination (SHE)

To reduce harmonic distortion, the modulating voltages V_d and V_q obtained from the MPC algorithm are adjusted in conjunction with the SHE technique. SHE is a widely-used method for eliminating specific harmonics by carefully selecting the switching angles of the inverter.

In this implementation, SHE works with the modulating voltages to ensure that problematic harmonics (e.g., the 5th, 7th, and 11th harmonics) are minimized while maintaining the desired output voltage waveform. The combination of MPC and SHE provides a robust harmonic mitigation strategy that surpasses the performance of either method alone.

Cost Function Optimization using Gradient Descent

One of the key challenges in MPC is solving the cost function that minimizes the tracking error between predicted and reference voltages while respecting system constraints. In this implementation, gradient descent is employed to optimize the modulating voltages.

Cost Function Formulation

The cost function J is designed as stated in equation 3.2

$$J[k] = \|i^*_\alpha[k+2] - i_{p,\alpha}[k+2]\|^2 + \|i^*_\beta[k+2] - i_{p,\beta}[k+2]\|^2 \tag{3.2}$$

3.1.4 Gradient Descent Optimization

Gradient descent is used to iteratively minimize the cost function. The modulating voltages are updated like in equation 3.3.

$$\begin{aligned} V_d[i+1] &= V_d[i] - \eta \frac{\partial J}{\partial V_d} \\ V_q[i+1] &= V_q[i] - \eta \frac{\partial J}{\partial V_q} \end{aligned} \tag{3.3}$$

Where i is the iteration index, η is the learning rate, and $\frac{\partial J}{\partial V_d}$ and $\frac{\partial J}{\partial V_q}$ are the gradients of the cost function with respect to V_d and V_q, respectively.

This approach ensures an efficient optimization process, suitable for real-time applications. Using this strategy of obtaining the correct modulating voltages also makes the system easily changeable in the code by adjusting a single parameter that controls the amount of prediction horizons.

For the implemented simulation, the according equation for the gradient descent function is derived from equation 3.2, and stated in equation 3.4.

When α components are predicted:

$$\begin{aligned} V_d &= V_d + \eta * 2 * (i^*_\alpha[I] - i_{p,\alpha}[I]) * T_s/L_d * \cos(\theta_e + (I+1) * \omega_s * T_s) \\ V_q &= V_q + \eta * 2 * (i^*_\alpha[I] - i_{p,\alpha}[I]) * T_s/L_q * sin(\theta_e + (I+1) * \omega_s * T_s) \end{aligned}$$

When β components are predicted:

$$\begin{aligned} V_d &= V_d + \eta * 2 * (i^*_\beta[I] - i_{p,\beta}[I]) * T_s/L_d * \sin(\theta_e + (I+1) * \omega_s * T_s) \\ V_q &= V_q + \eta * 2 * (i^*_\beta[I] - i_{p,\beta}[I]) * T_s/L_q * \cos(\theta_e + (I+1) * \omega_s * T_s) \end{aligned} \tag{3.4}$$

Where V_d and V_q are updated for every prediction of either an α or β component. T_s represents the sampling time or updating time of the MPC controller, θ_e is the electrical motor angle and I denotes the entry in the array, where the last information of the last or highest horizon is stored. The distinction between the sets of equations for α and β components is important, as it stems from the fact that the inverse Park transformation is involved. For

implementing a correctly functioning code, it should also be noted that the V_d and V_q components can not instantly be updated after either a new α or β components is calculated, due to the fact that the cost function will not have the same input V_d and V_q variables for both α and β components.

3.1.5 Comparison with PI Control

A property of this implementation is the ability to directly compare the MPC-generated V_d and V_q voltages with those obtained from a traditional PI controller. The MPC boasts superior tracking performance, particularly during rapid reference changes, as it considers future behavior over the prediction horizon. PI controllers tend to have slower transient responses, because they work with information from the present and past.

3.2 Implementation of the Selective Harmonic Elimination (SHE) Technique

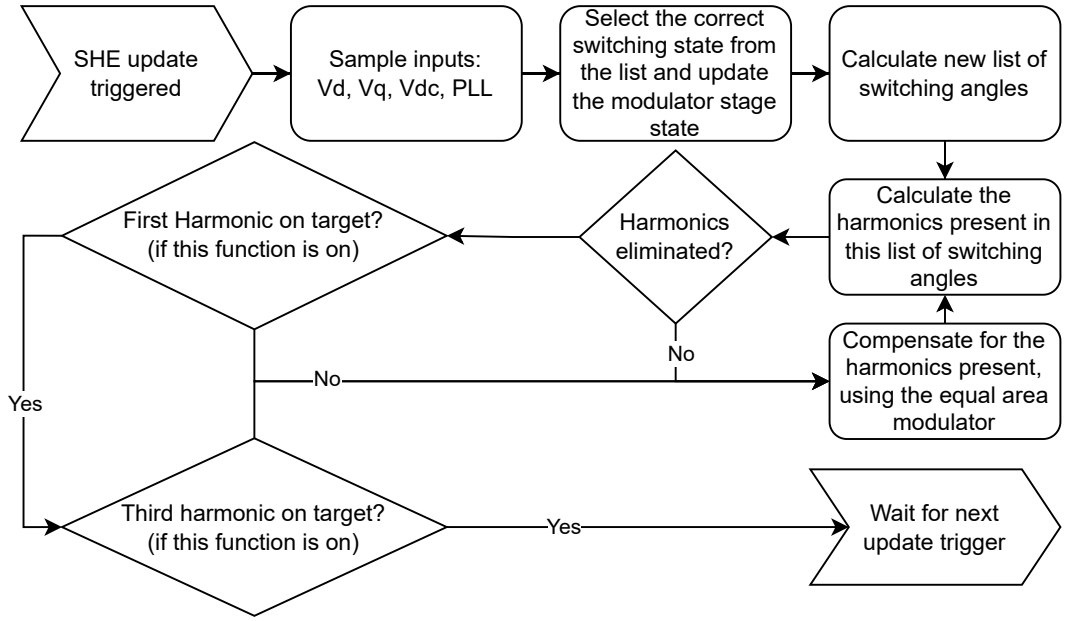


Figure 3.1: Flow chart of the SHE algorithm

As mentioned previously, SHE is a useful technique for improving the harmonic contents of output waveforms in power electronics. This technique is chosen for several compelling reasons that make it more applicable to other methods.

3.2.1 Precision in Harmonic Elimination

The primary motivation for using the SHE technique is its precision in eliminating specific harmonic components from the output voltage waveform. By accurately calculating and compensating for the harmonic content through iterative processes involving Fourier series analysis, SHE ensures that undesired harmonics are effectively minimized. This precision leads to a cleaner output. Reducing lower order harmonics using SHE can increase the higher order harmonic components, however for applications like power grids, this is advantageous. Because the lower order harmonics are reduced, a smaller filter size can be utilised, leading to a possibly more efficient output and lower filter cost that meets stringent power quality standards.

3.2.2 Efficiency of Iterative Processes

The iterative nature of SHE, incorporating Fourier calculations and compensation using the equal area criteria, allows for on-demand adjustment of the switching angles. A flow chart of this iterative process is presented in Fig. 3.1. This flexibility contrasts sharply with traditional SHE techniques that rely on precomputed lookup tables. In those methods, the fixed nature of the lookup tables limits adaptability and responsiveness to changes in operating conditions. SHE's iterative approach, however, can dynamically adapt to varying conditions, ensuring optimal performance without the need for extensive precomputed data.

The equal area modulator only calculates θ_k's, which is the switching angle at which the modulator goes to the on state. However, an additional mode was implemented, where the σ_k's can also be compensated. The derivation for this leads to just flipping the input variables in equation 2.17, where σ_k becomes σ_{k-1} and vice versa. After the iterative SHE process is ran, it may occur that the amplitude of the fundamental changed. To fix this, the fundamental is also taken into account in the iterative process, like stated in Fig. 3.1. The goal is then not to eliminate the first harmonic, but to eliminate the deviation from what it originally was when the equal area modulator was calculating a fresh set of switching angles. This process can also be repeated for the third harmonic, to obtain THI. This functionality is also present in the code, which can ofcourse be seen in section D.

3.2.3 Universality and Flexibility

Another significant advantage of this SHE technique is its universality. Unlike some other nonlinear SHE methods that require solving complex nonlinear equations specific to a given application, this technique provides a universal framework. This means it can be quickly altered and applied to different applications without extensive reconfiguration. The ability to generalize SHE across various applications enhances its practicality and efficiency, making it a versatile tool in the field of power electronics.

3.2.4 Implementation of the Fourier series computation

To compute the calculations for this part, edge detection on the phase to neutral voltage is done for all of the three phases. All these switching timestamps, along with the voltage level of the peak or zero state, are saved for calculating the harmonic constants of a period later on. When the electrical motor angle reaches 2π, the Fourier constants are calculated by iterating over the saved timestamps with their according voltage level and inserting them into equation 2.10. Obtaining accurate period detection is crucial for the SHE algorithm to operate correctly and care must be taken to avoid fluke period detections. This Fourier calculation is then used to check and evaluate the effectiveness of the SHE algorithm, but this calculation algorithm can also be used for the first step of the SHE compensation algorithm with slight modifications. For the SHE calculations it is imperative that the switching timestamps of the next period are calculated and compensated for to eliminate the targeted harmonic components. This is done by first computing a set of switching angles that result in a correct first harmonic level (the fundamental), by using the equal area modulator. After the switching angles are found, the Fourier calculation is performed for the coming period and the resulting non-zero harmonics can be eliminated using the equal area modulator again. Again, the whole process can be reviewed in Fig. 3.1.

3.2.5 Implementation hurdles

All SHE methods rely on accurate modulating, where the states are switched at precisely calculated angles or phases. Because of the inherent nature of SHE to have a relatively low amount of switching angles per electrical period, the problem can arise where the motor speed starts to oscillate correlated to the calculated switching angles. This oscillating behaviour of the motor speed has its effect on the electrical phase as well, because it is determined from the motor speed by integrating it. The modulating wave forms of the three phases are also influenced by this effect, which thus works against the proper functioning of the SHE algorithm.

3.2.6 Oscillation improvement idea using sign alternation

The previously described problem was thought to be caused by on and off times being too long in comparison to the inertia of the machine. This is analogues to the PWM frequency being too low for the machine. The equal area modulator starts a period in a fixed state (the low state). It calculates when the phase voltage should jump to the high state. If this order were reversed, the hypothesis can be made that the effects that the normal period will have on the machine, would be cancelled out or reversed by the alternate period, where the start sign of the equal area modulator is different. Implementing this supposed fix for the motor oscillations requires the formula of the equal are modulator to be altered. Equations 2.17 and 2.19 contain the default equal area modulator and running through the derivation for them again, with the addition of the first sign being altered, results in equations 3.5 and 3.6.

$$\theta_k = \frac{\sigma_k}{2} + \frac{\sigma_{k-1}}{2} + \frac{A(cos(\sigma_k) - cos(\sigma_{k-1}))}{V_{DC}} \tag{3.5}$$

$$\theta_k = \theta_{k,\text{previous iteration}} + \sum_{3}^{5,7,11...} \frac{\frac{h_m}{m}(cos(\sigma_{k-1}) - cos(\sigma_k))}{V_{DC}} \tag{3.6}$$

It can be seen that these equations are similar to equations 2.17 and 2.19, where the only difference is that σ_k is substituted for σ_{k-1} and vice versa. However, the results of this change show that this does not improve the situation and that the system becomes even more unstable. This unstable behaviour was thought to arise because of the fact that at the instance when the sign is flipped, the modulator output state stays fixed for a moment, before continuing to the other state, which completes the sign alternation. This sign alternation can thus amplify the instability in that instance, as the motor speed change continues during the sign alternation, which could make the system even more unstable, removing the possible good effects of this technique.

3.2.7 Designing a Phase Locked Loop for accurate phase detection

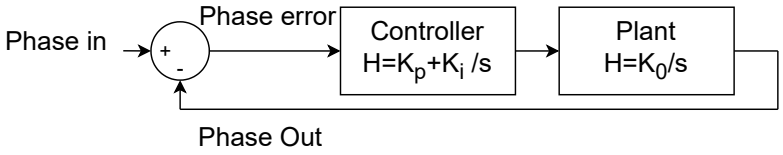


Figure 3.2: The control loop of the PLL

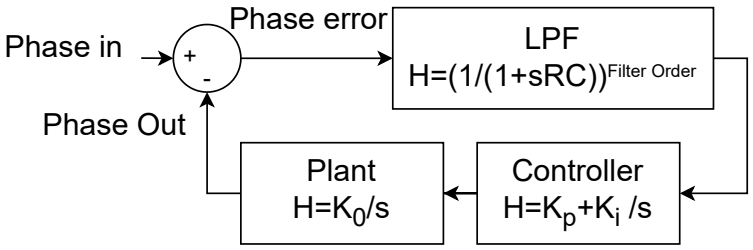


Figure 3.3: The control loop of the PLL with the added low pass filter

Another solution could be to implement a phase locked loop. Thus, a phase locked loop is to be designed, which can filter out the unwanted oscillations to determine a useful phase signal for the SHE algorithms to work with. The proposed PLL is pictured in Fig. 3.2. A proportional-integral (PI) controller is used to achieve correct tracking of the reference signal. The PI controller can be tuned to a certain cutoff frequency, determining a suitable value for this is crucial. The plant is simply an integrator with a gain (K_0) of one, because a stepped input to the plant results in an output of one after a second. Additionally, the input of the plant and the output of the controller represents the electrical rotor speed.

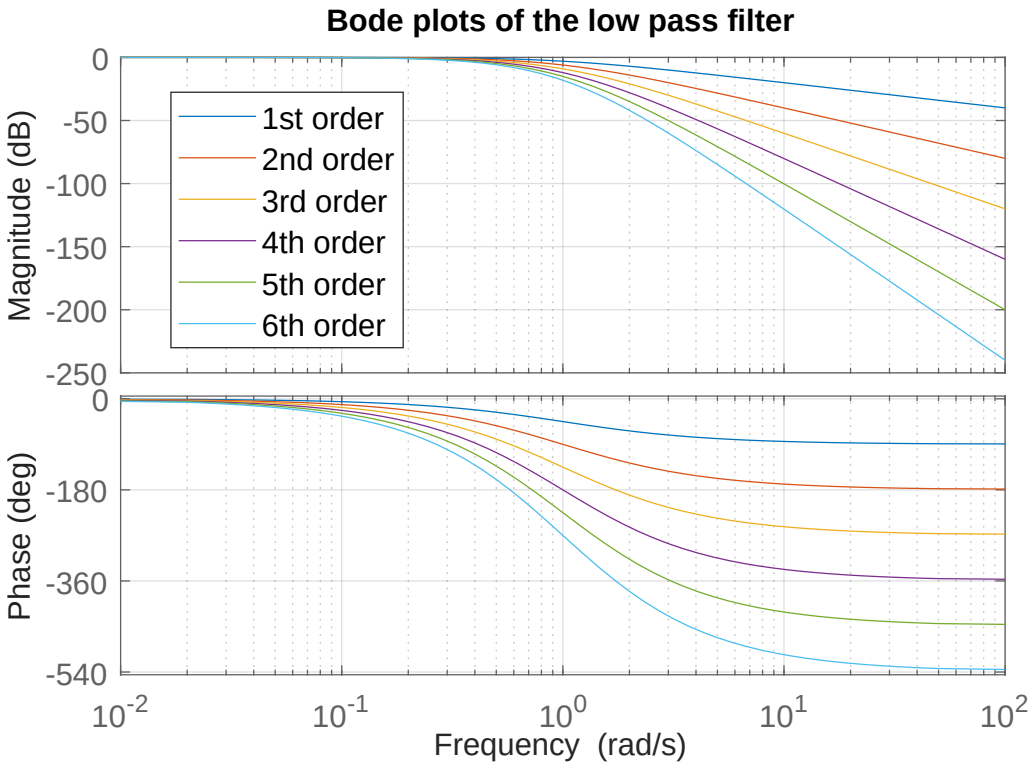


Figure 3.4: Bode plots of the low pass filter for orders one to six

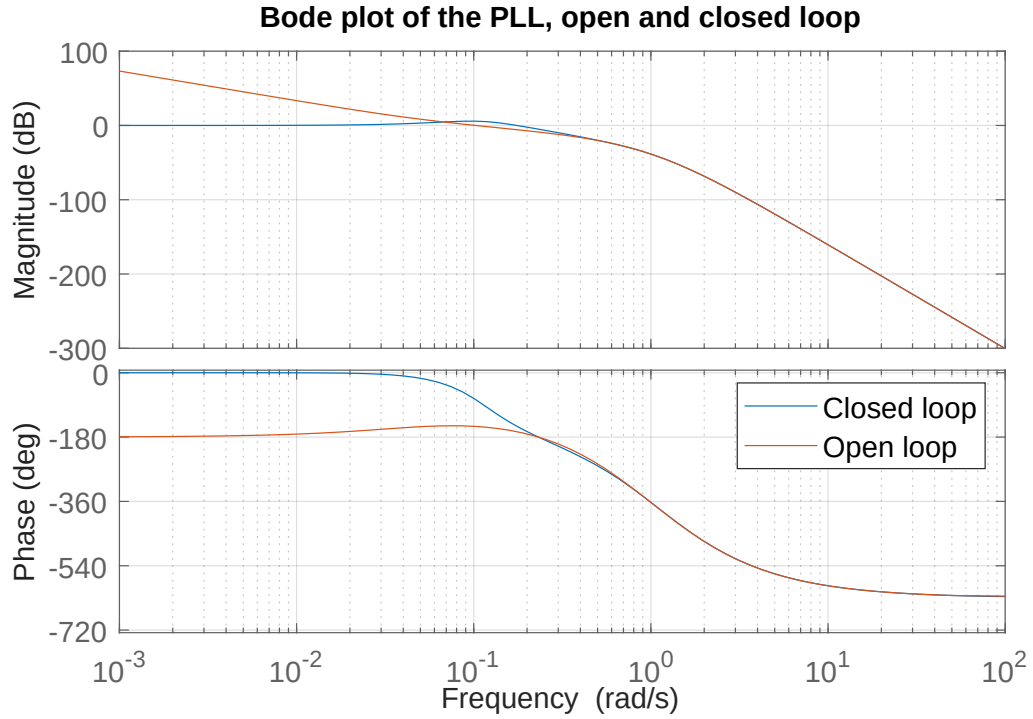


Figure 3.5: Bode plots of the PLL control loop in closed loop and open loop form

In order to further improve the performance of the PLL, an additional digital low-pass filter was added, to further decouple the PLL from disturbances and oscillations. The modified block diagram of the control loop can be seen in Fig. 3.3. This low-pass filter is a 6th order cascaded filter. Making the filter one of the 6th order ensures quick roll-off behaviour, while having relatively little phase delay at the cut off frequency, as it was noticed that going to higher order filters, results in the cut-off frequency not significantly reducing. In Fig. 3.4 a bode plot is presented for the different filter orders. In the implementation, it is an option to have the filter cut-off frequency and PI cut-off frequency be calculated by the code in real time and adjust the filter and PI parameters. Depending on the motor speed and the PWM frequency or number of switching angles per period, the filter and PI parameters are updated. Since the effective PWM frequency introduces the most noise in the motor speed, the PLL is usually operated at that frequency with a target gain of for example -40 dB. When SHE is operating, a specified number of switching angles per period are present. The effective PWM frequency for the real time updating of the parameters is then calculated to be: $N_{switching\ angles\ per\ period}/2 * \omega_{synchronous}$. Additionally, another feature is also present in the code, where the response of the PI can be adjusted so that a certain phase margin is maintained. The PI and plant of the PLL are analysed open loop, which results in a phase margin for a certain cut-off frequency of the PI. This phase margin is calculated to be 66°. The code then uses simple interpolation between the cut-off frequency and a decade less than that, to obtain a simple representation of the phase delay. A value in the code can be set that subtracts a little from the phase margin to use for the filter stage. In the case of Fig. 3.5, this value was set to -6°. From Fig. 3.5 it can be seen that around 30° of phase margin is left in open loop state, which means that the interpolation was not entirely accurate, but it should be accurate enough to make the PLL operate nicely. If needed, the value which controls the allowed phase margin can be set higher to obtain more phase margin for the whole system. To calculate the values for K_p and K_i, equation 3.7 was the starting point and equation 3.11 gives the resulting PI values.

$$|C * P|_{\omega=\omega_{bandwidth}} = 1 \tag{3.7}$$

$$|\frac{K_p * s + K_i}{s} * \frac{K_0}{s}|_{\omega=\omega_{bandwidth}} = 1 \tag{3.8}$$

$$|K_p * j\omega_{bandwidth} + K_i| = \frac{\omega^2_{bandwidth}}{K_0} \tag{3.9}$$

$$K_p^2 = \frac{-\omega^2_{bandwidth} + \sqrt{(}\omega^4_{bandwidth} + \omega^4_{bandwidth})}{0.5 * K_0^2} \tag{3.10}$$

$$\begin{aligned} K_p &= \frac{\omega_{bandwidth} * \sqrt{(}2(\sqrt{(}2) - 2)}{K_0} \\ K_i &= \frac{\omega^2_{bandwidth} * (\sqrt{2} - 1)}{K_0} \end{aligned} \tag{3.11}$$

The closed loop transfer function, a long with the open loop transfer function can then be modeled in MATLAB, and Fig. 3.5 depicts the results. The low pass filter is characterised

as follows from equation 3.12. The associated formulas are presented from the information available in [20].

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1} \tag{3.12}$$

In equation 3.12 a new term α arises, which is defined in equation 3.13.

$$\alpha := \frac{T_s}{RC + T_s} \tag{3.13}$$

From the expression for the cut-off frequency of the filter (f_c) in equation 3.14 the equation for RC can be determined.

$$f_c = \frac{1}{2\pi RC} \quad \text{so} \quad RC = \frac{1}{2\pi f_c} \tag{3.14}$$

The presented equations are what is needed to implement the filter in a code based solution. The written code for the PLL can also be read through in section C.

3.2.8 Controller of FOC

(In normal foc) The controller also benefits from filtering, as with high speed operation, the current waveforms start to look more 'jagged' as the triangular nature of the motor inductor currents become more apparent. This can be seen in the default model which was provided in the PLECS example library. The Id and Iq currents are very noisy, because of this reason, thus providing filtering like achieved previously in section 3.2.7, would reduce the noise.

3.3 Dissipating energy in the system

In PLECS a controlled torque source is used, which has a negative torque value, thus actually becoming a torque 'sink'. The benefit of using this component is that the motor speed will stay relatively constant, as the torque that the motor produces is equal to the torque that the sink produces in the opposite direction. Then a net effective torque close to zero is applied to the motor inertia, which means that the speed does not increase or decrease, but stays constant.

3.4 Implementation of the LMS-based SHE

This SHE method was implemented in PLECS as well, according to the block diagrams depicted in Fig. 3.7, 3.8 and 3.9. This can be seen in Fig. 3.6. The previously discussed PLL can be recycled for use with this C-Script and is present on the second input of the code block. Additionally, the stator currents are needed for this scheme and they are present on the third input. The tracking of the harmonic components is done on the three phase current signals. The LMS algorithm then does its task and the output of the code block is present

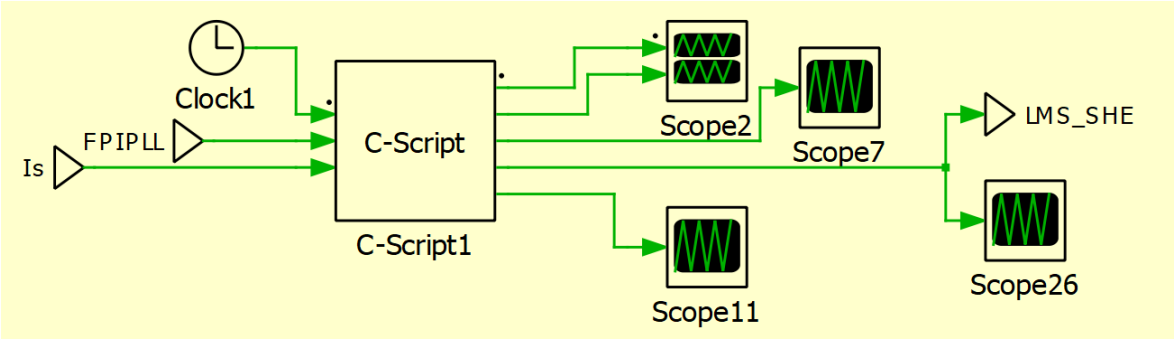


Figure 3.6: The schematic of the implementation of the LMS-based SHE method in PLECS

on the LMS_SHE signal port in PLECS. These signals can then be superpositioned onto the three phase modulating waveforms which go into the SPWM block of the model.

To predict the back emf for every horizon, the code excerpt 1 used in the MPC implementation, presents the solution. The rest of the code used for the MPC controller can be found in E.

Algorithm 1 This code excerpt shows the generation of input variables to the model predictor for every horizon. p.Uemf[i][j] is the two-dimensional array where the U_{emf} for α and β are stored. The integer i refers to α and β when it is equal to zero and one respectively. Integer j is the variable that specifies which horizon is calculated. The InputSignal function retrieves the amplitude of the back emf, which is calculated outside the C-block by multiplying the synchronous speed by the permanent magnet flux linkage.

```
if(i==0) {
    p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*-sin(M.angle+j*M.speed*Ts);
    p.U[i][j] = invParkAlpha(p.Vd, p.Vq, M.angle+j*M.speed*Ts);
    p.iGoalFurthestHorizon[i] = invParkAlpha(p.idGoal, p.iqGoal, M.angle+
        ↪ PREDICTIONS*M.speed*Ts);
} else {
    p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*cos(M.angle+j*M.speed*Ts);
    p.U[i][j] = invParkBeta(p.Vd, p.Vq, M.angle+j*M.speed*Ts);
    p.iGoalFurthestHorizon[i] = invParkBeta(p.idGoal, p.iqGoal, M.angle+
        ↪ PREDICTIONS*M.speed*Ts);
}
```

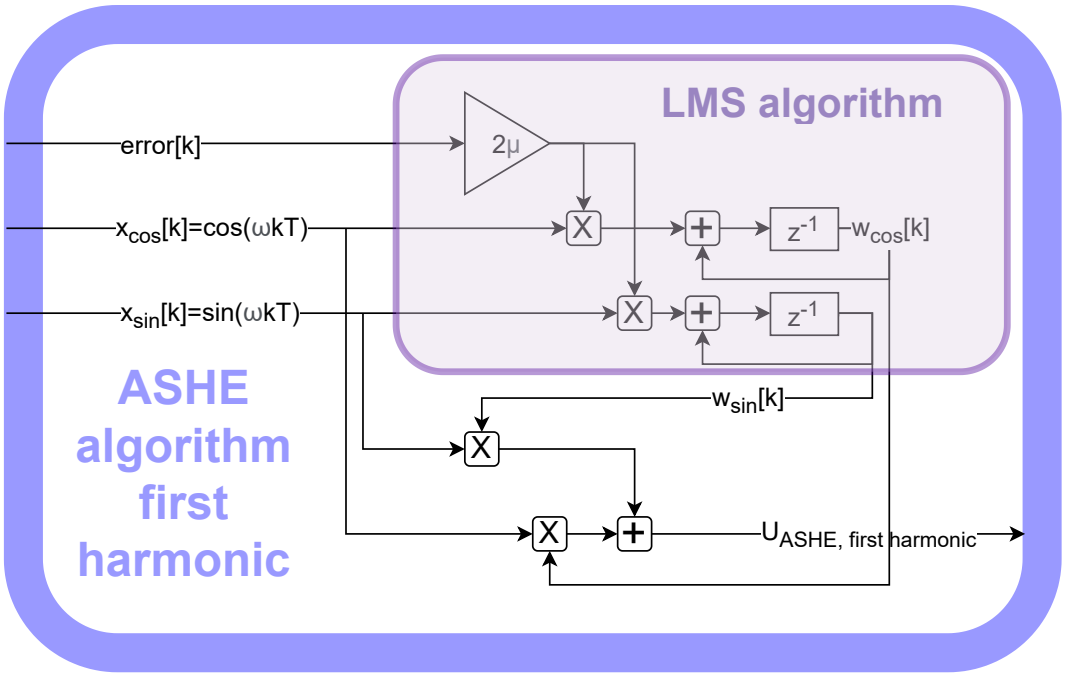


Figure 3.7: The block diagram of the ASHE algorithm that is used for tracking the amplitude of the first harmonic

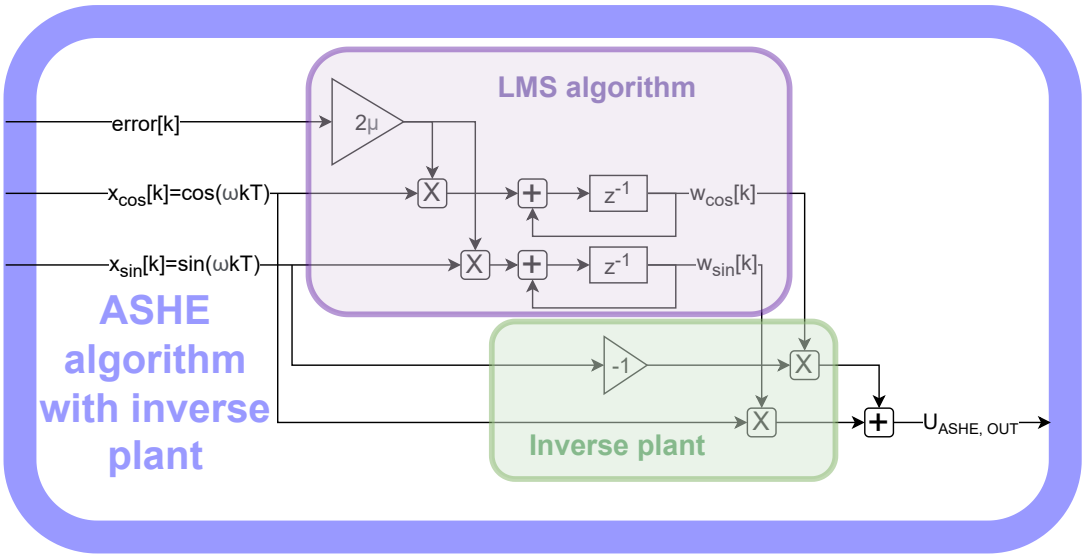


Figure 3.8: The block diagram of the ASHE algorithm that is used for generating the voltages that are added to the modulating signals

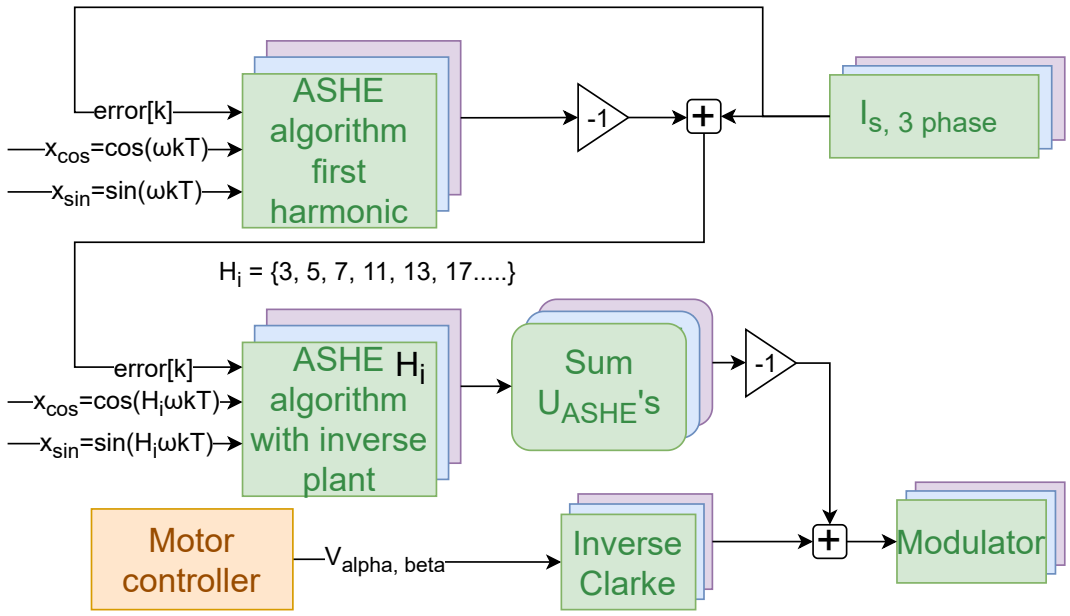


Figure 3.9: The block diagram of the implementation of the LMS-based SHE method

Chapter 4

Results

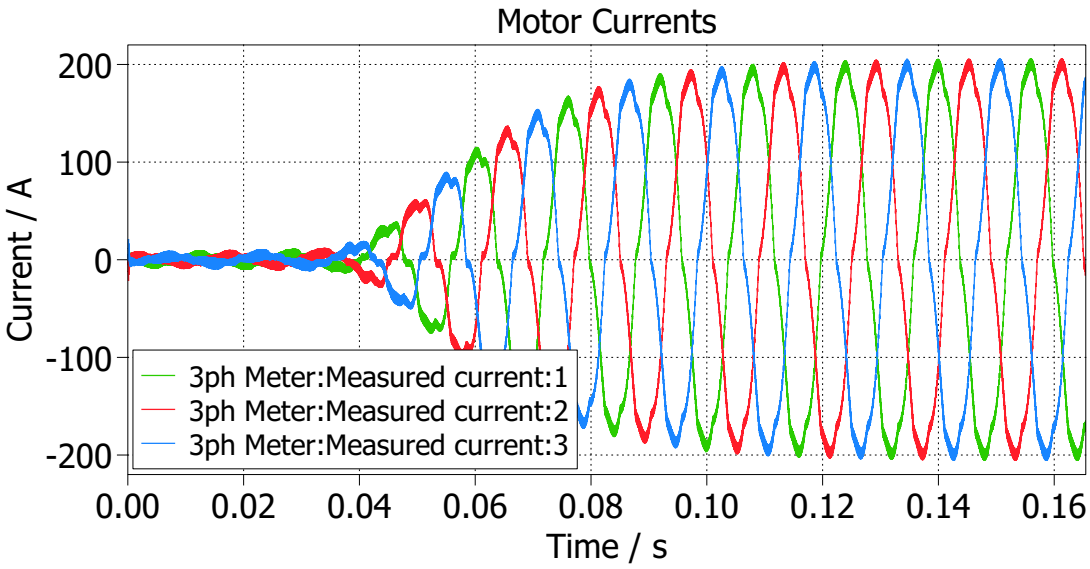


Figure 4.1: Phase currents startup with the PI carrier-based implementation

The modeled motor and connected components in PLECS have the specifications stated in table 4.1. The motor parameters are inline with [18] and [21]. Additionally, [22] presents a high-power motor with significantly less stator inductance. It is interesting to notice that certain motor parameters can vary widely, while the characteristics of the motor power are the same. To provide an overview, in table 4.2 the frequencies at which parts of the system operate are given. The cut-off frequencies of the PLL where obtained by setting the filter target to be 40 dB attenuation at 2 kHz in the code, which can be retrieved from section C.

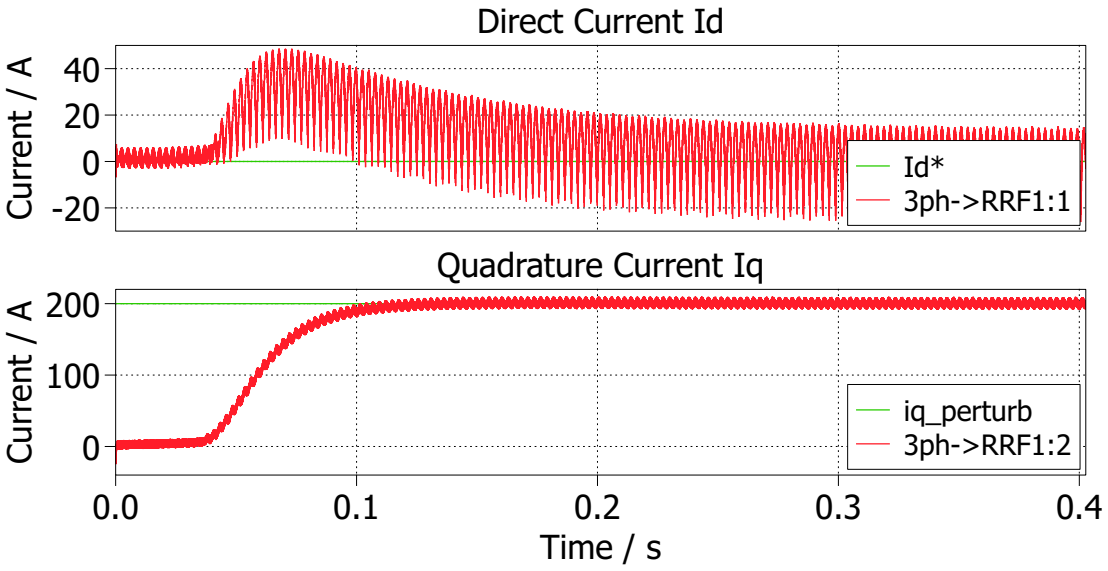


Figure 4.2: Direct and quadrature currents from 0 to 0.4s simulation time

Table 4.1: Hardware properties of the simulated system

Motor inductance (L_d, L_q)	200 μH
Stator resistance (R_s)	20 mΩ
Motor poles	8 (4 pairs)
DC link voltage	400 V
Permanent Magnet Flux Linkage	0.15 Wb

Table 4.2: Frequencies at which system parts operate

PI-based FOC controller bandwidth	20 Hz
PWM frequency	10 kHz
MPC sampling frequency	100 kHz
LMS-based SHE sampling frequency	50 kHz
SHE update frequency	10 kHz
Fourier components calculation update frequency	Continuous (when new period is detected)
PLL update frequency	100 kHz
PLL cut-off frequency of the PI	97 Hz
PLL cut-off frequency of the filter stage	928 Hz

4.1 PI-based controller with no SHE

To provide a baseline, the system is simulated in a bare-bones way, where the controller is a PI-based one with SPWM modulation set to a bandwidth of 20 Hz. The target current quadrature current is set to 200 A and zero direct current is required. The waveforms of the stator phase currents and the I_d and I_q are viewable in Fig. 4.1 and Fig. 4.2 respectively. The PWM frequency was set to 10 kHz and the mechanical speed of the motor at 100 rad/s.

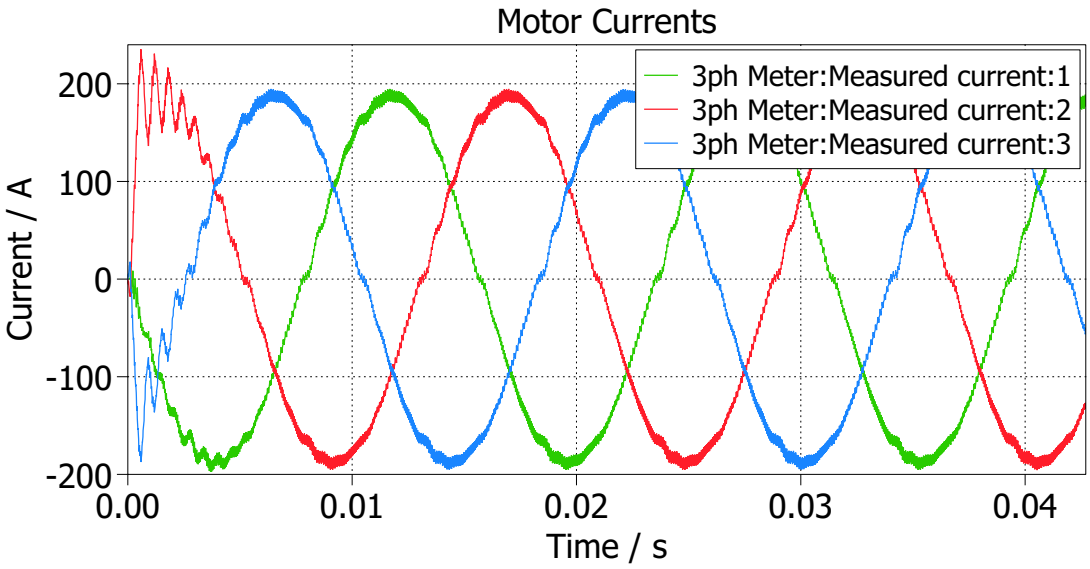


Figure 4.3: Phase currents with the carrier-based indirect MPC connected

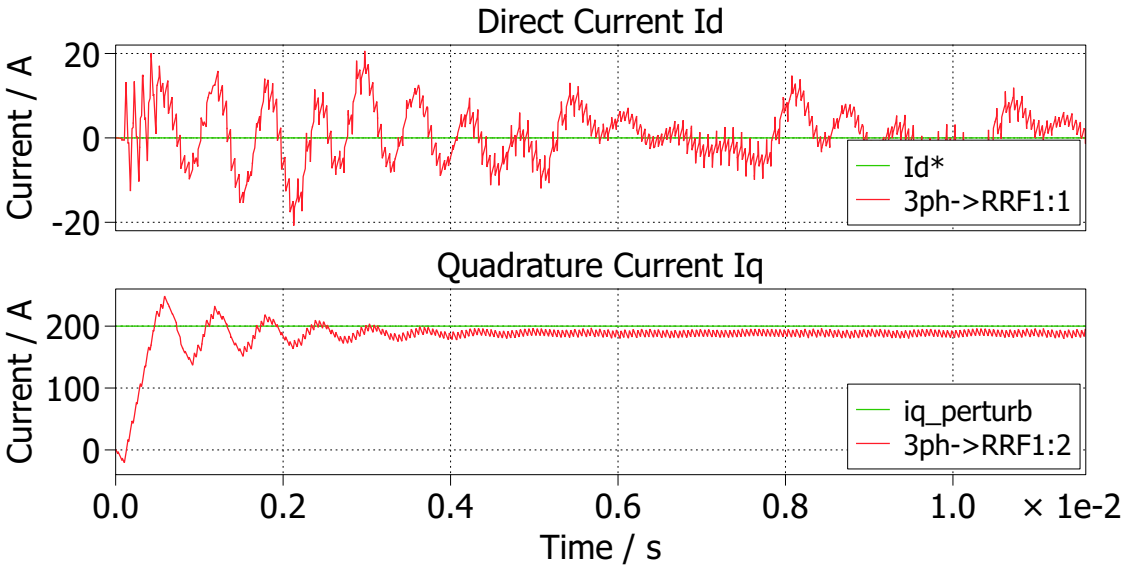


Figure 4.4: D and Q currents with the carrier-based indirect MPC connected

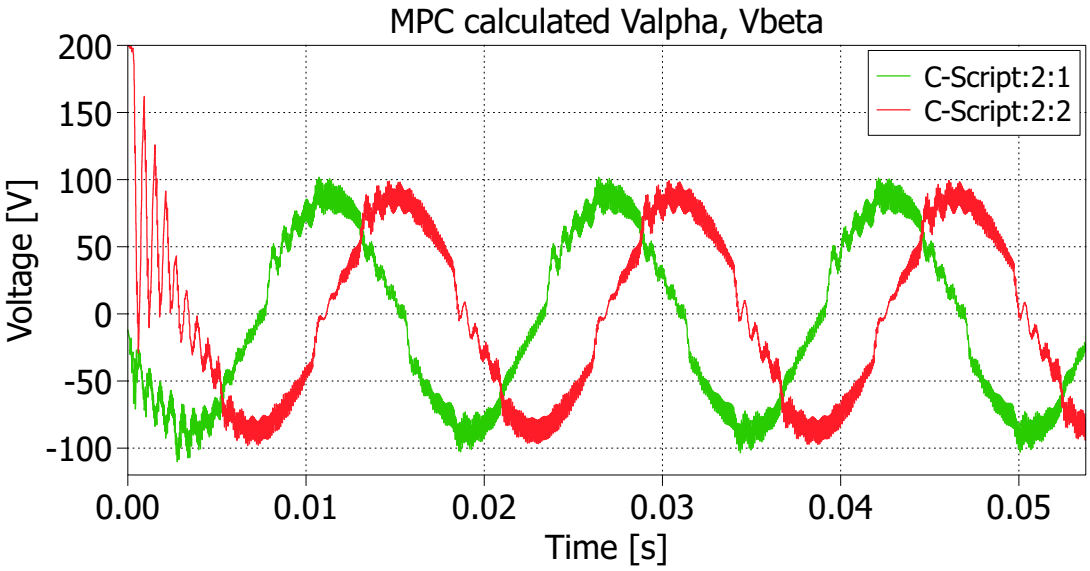


Figure 4.5: Calculated V_α and V_β by the MPC

4.2 Indirect MPC

Connecting the discussed indirect MPC controller to the motor simulation results in the i_d and i_q waveforms present in Fig. 4.4. Additionally, the stator phase currents can be seen in Fig. 4.3. The currents are not tracking their references however, this means that some disturbance observer or a separate compensator with integrator behavior needs to be implemented to combat this behavior. However, compared to the original PI controller, the currents rise to their targets in an instant of time. What can also be noticed is that there is slightly less current ripple in the D and Q reference frame with the MPC implementation. The MPC calculates the V_α and V_β correctly, as can be seen in Fig. 4.5. There is ripple present, which be attributed to the fact that the carrier based PWM does not update instantly, so the MPC is correcting for a change that did not happen yet.

The MPC controller itself already has superior performance in terms of harmonic elimination, but of course the MPC controller does not directly incorporate harmonic elimination. THD was also improved compared to the PI-based FOC controller. The results are also added to table 4.3 and the harmonic components are normalized to the 200 A current target, as the other entries in the table were operating at that setpoint and the MPC's current measured showed slightly lowered values.

It is important to set an appropriate learning rate for the MPC controller. In this simulation a learning rate (η) of ten provided quick convergence of the required modulating voltages. Additionally, the cost was optimized to a value of 0.005 or below to provide accurate results. As mentioned before, eleven horizons are predicted and the amount of allowed iterations was set to a maximum of 100. Setting this maximum number of iterations can be important, as otherwise the code can never solve and meet the cost criteria, which would result in an infinite loop. The amount of iterations can also be used in a system where the learning rate (η) is set by some controller in the program itself, which tries to match a specified amount of

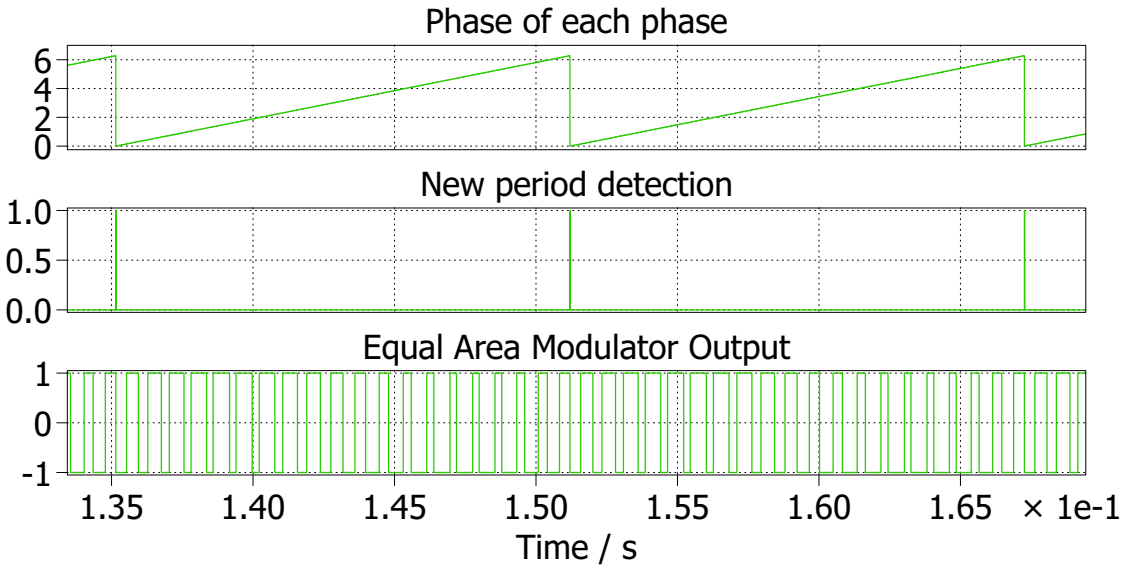


Figure 4.6: Shows the correct calculation of the switching angles. In this case SHE is turned off.

iterations to reach convergence.

4.3 SHE correct operation check

In Fig. 4.6 the correct operation of the equal area modulator can be observed. The graph shows the modulated signal for a full period of a single phase. Only the equal area modulator is active in Fig. 4.6, which is equal to the first step in the process of Fig. 3.1. To speed up the process, the list with ”old” switching angles can be used directly, when the compensation is on for the first harmonic. Fig. 4.7 shows the modulated signals for the case when SHE is turned on. The waveform looks similar to Fig. 4.6, which is ofcourse logical, as the equal area modulator should only correct for the harmonics, which does not change the shape of the modulated waveform significantly.

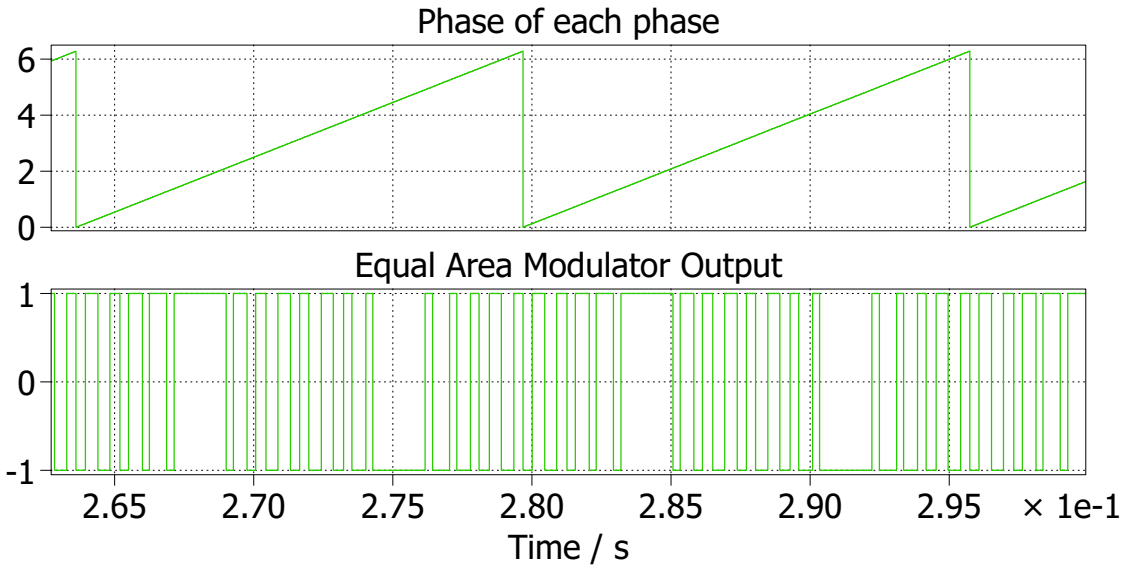


Figure 4.7: Shows the correct calculation of the switching angles. In this case SHE is turned on.

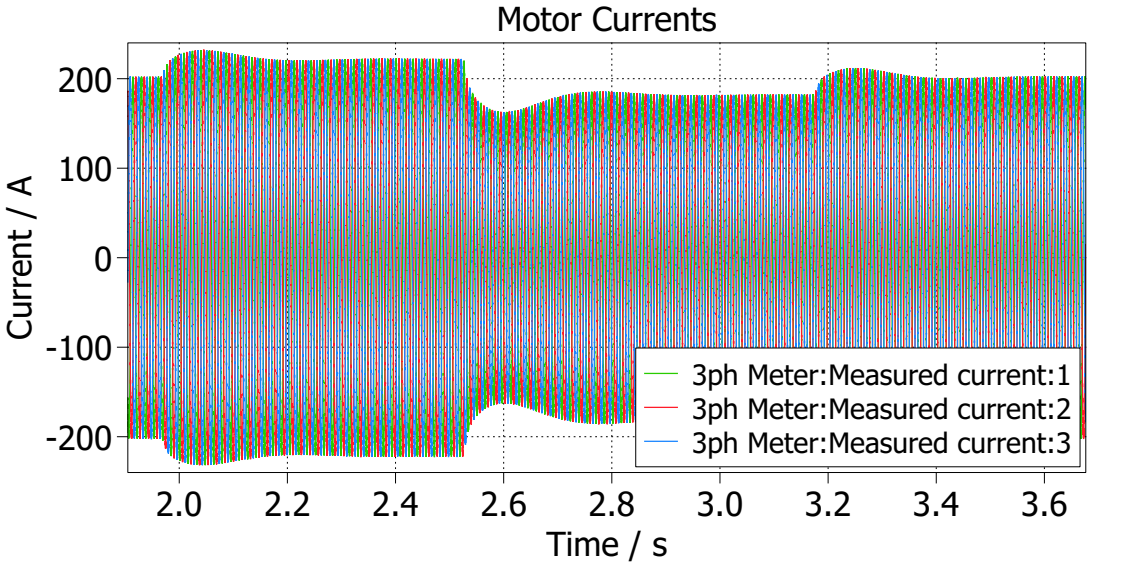


Figure 4.8: Shows that the LMS algorithm can handle changes of the Q current setpoint

4.4 PI-based FOC controller in combination with SHE

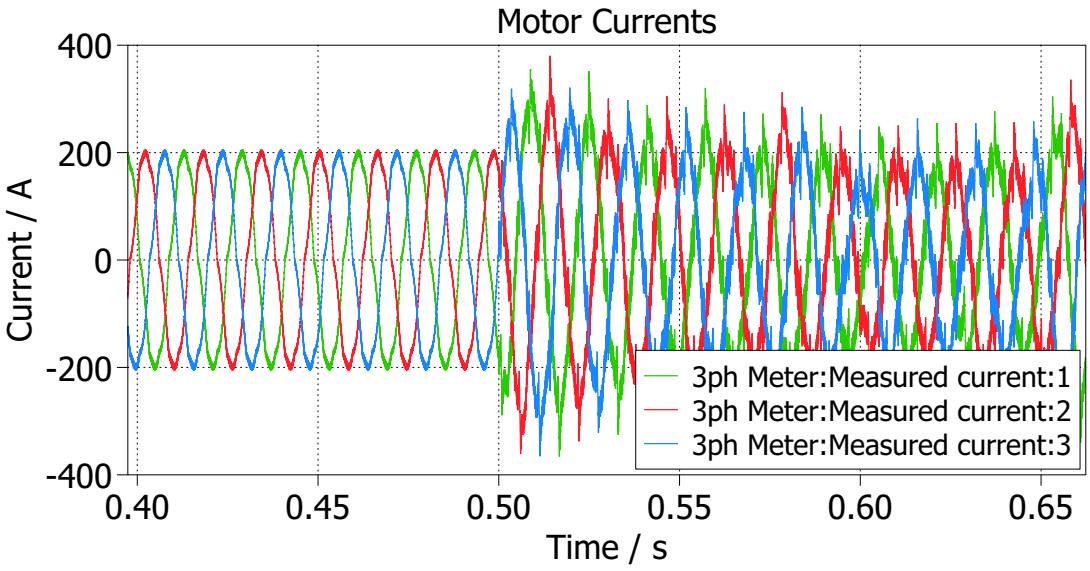


Figure 4.9: Stator phase currents with the SHE algorithm operating from 0.5 s onwards. 300 switching angles were used in this instance to compare with the 10 kHz PWM frequency

The simulated output stator currents and dq currents pictured in Fig. 4.9 and 4.10 reflect that the system is unstable. The PLL does its job in terms of filtering away fast transients in the motor control (Fig. 4.11), which can reflect onto the phase of the three phase currents. However, the addition of the PLL was not enough to achieve a steady current signal. Additionally, low-pass filtering of the required V_d and V_q signals was tried as a solution to the unstable currents by adding a moving average filter. This did not yield an improved result, however, which means that this SHE method is less suitable for use in this specific application. The resulting Fourier spectra show that the harmonics have not reduced. To get a sense of what could be the issue at hand, the modulated waveform was passed onto an RL-circuit in PLECS, with the same L_{stator} and R_{stator} values as in the main simulation. It was noticed (Fig. A.1 in section A) that putting these modulated voltages onto the RL-circuit also introduces peaky behavior in the amplitude or maxima of the current waveform. The reason could be that in the carrier based modulator, the gating signals (Fig. A.2 in section A) are ”synchronized”, meaning that they overlap most of the time. The output of the equal area modulator, which is visible in Fig. A.3 in section A, does not have this inherent synchronization, meaning that this unwanted behavior can result.

4.5 Indirect MPC combined with SHE

The system is operational with the MPC connected to the SHE algorithm, however the quality of control is not there, similar to the situation where the PI controllers were used. Thus

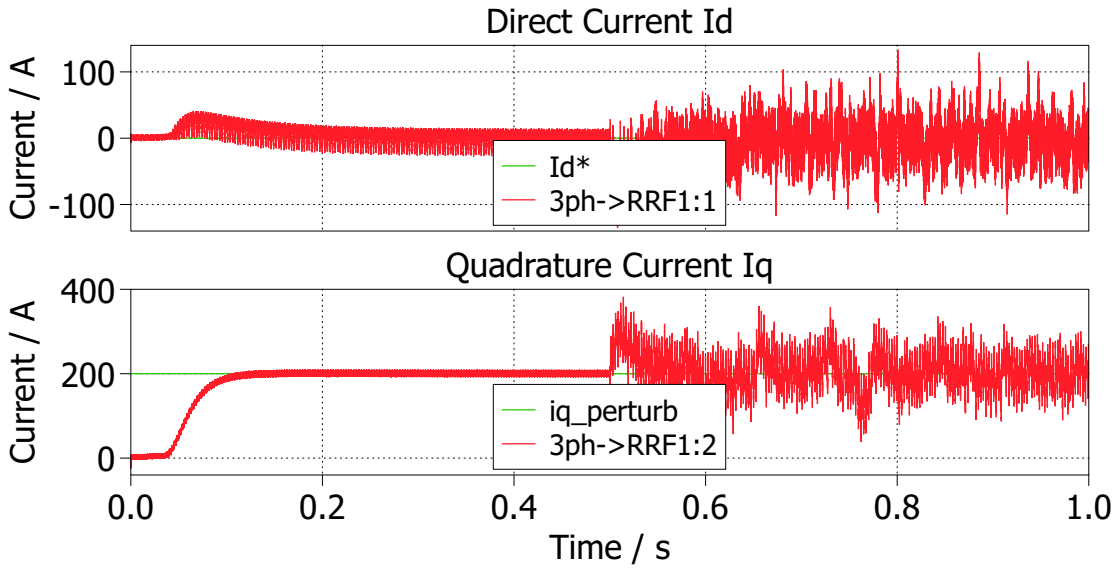


Figure 4.10: D, Q currents with the SHE algorithm operating from 0.4 s onwards

the inclusion of MPC did not improve the previously mentioned behavior. One thing that is important to notice, is that the MPC approach of SHE can operate the motor better when lots of current ripple is present or the synchronous frequency approaches the PWM frequency.

4.6 LMS-based SHE operation validity check (ASHE)

The simulated output of the individual LMS algorithm compensation components can be seen in Fig. 4.12. It was noticed that the waveforms increase in amplitude gradually over time, owing to the principle of operation behind the LMS algorithm, which confirms that the system is operating appropriately.

4.7 PI-based FOC controller in combination with LMS-based SHE

Table 4.3: The results of the LMS-based SHE compared to the PI-based FOC without SHE

	PI baseline	LMS SHE after one second	LMS SHE after two seconds	MPC without SHE
THD	0.0583	0.047	0.040	0.029
5th harmonic amplitude [V]	8.7	6.2	4.6	2.4
7th harmonic amplitude [V]	6.2	5.6	4.4	1.8
11th harmonic amplitude [V]	2.0	1.3	2.3	1.3

The fifth and seventh harmonics were targeted to be reduced and looking at the Fourier spectrum graph yields a positive result. From Fig. 4.13 it can be seen that the fifth and seventh harmonics have reduced after the LMS algorithm became operational.

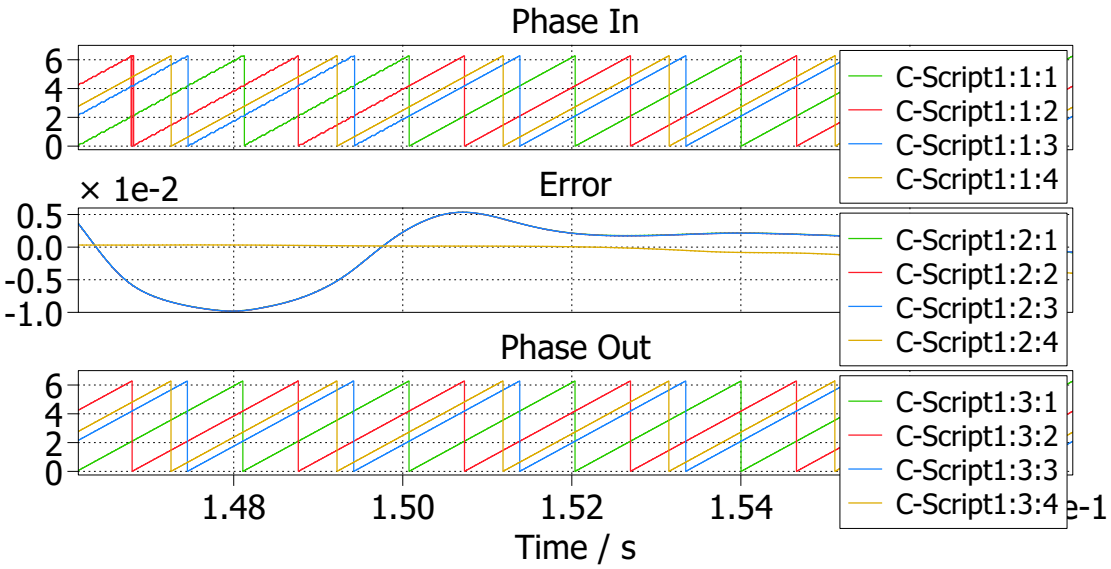


Figure 4.11: Graphs that display the correct operation of the PLL

After some heuristic testing of the LMS algorithm and adjusting the μ gain, a value of 0.0006 was best suited. The final results of the LMS-based SHE compared to no SHE method active is presented in table 4.3, where the simulation was run at a mechanical speed of 100 rad/s. From this table, it is clear that the LMS algorithm is able to reduce the harmonic content of the three-phase stator currents. The LMS algorithm works by detecting the harmonics in the current, this differs from the other SHE method, where the harmonics in the modulator voltage waveform are reduced. This difference means that the LMS algorithm can compensate for harmonics that resulted from the effects of dead time. This is an important distinction to make and depending on in what system SHE is needed, the ability of LMS-based SHE to compensate for harmonics originating due to the dead time, can be beneficial. The LMS gain is dependent on the simulation parameters and needs to be adjusted for different rotor speeds and motor parameters. Also a sampling frequency of 50 kHz was used for the LMS algorithm. Changing the sampling frequency also requires changing of the μ.

When THD is reduced in a system, the efficiency can possibly improve. In this case with the LMS-based SHE, the efficiency difference between using the LMS method and regular FOC came out to be 0.02%. Which means that there is a negligible difference in efficiency.

The dynamic performance of this scheme was also tested briefly. From Fig. 4.8 it can be seen that the changes in Q current setpoints are reached. The overshoots happen, because of the response of the PI controller, which also happen without the LMS algorithm attached. The sequence used here started with the Q current target set at 200 A then it was set to 220 A, 180 A and finally to 200 A.

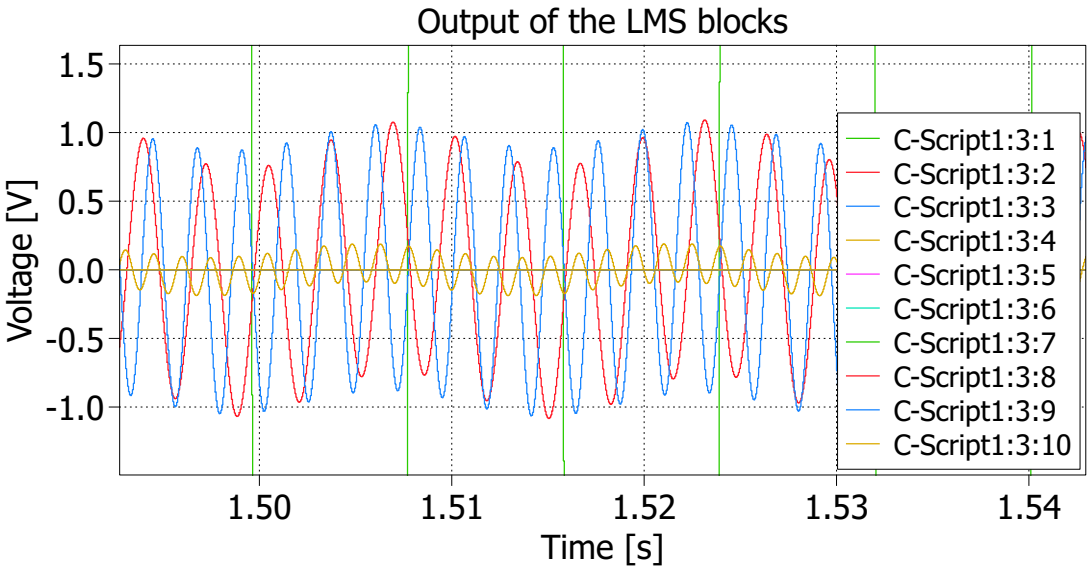


Figure 4.12: Injection signals referring to the LMS-based SHE method

4.8 Indirect MPC combined with the LMS-based SHE

Combining MPC and LMS did not provide further harmonic reduction. As can be seen from Fig. 4.14, the weights of the LMS algorithm keep increasing. This means that the LMS algorithm is not actually affecting the system, as otherwise the weights would stop increasing and reach a certain fixed amount. The MPC operates at a similar frequency to the LMS-based SHE, which means that it can compensate the changes the LMS-based SHE wants to make away. Incorporating a filter on the MPC inputs or outputs may be a viable solution to stop the effects these two controllers have on each other.

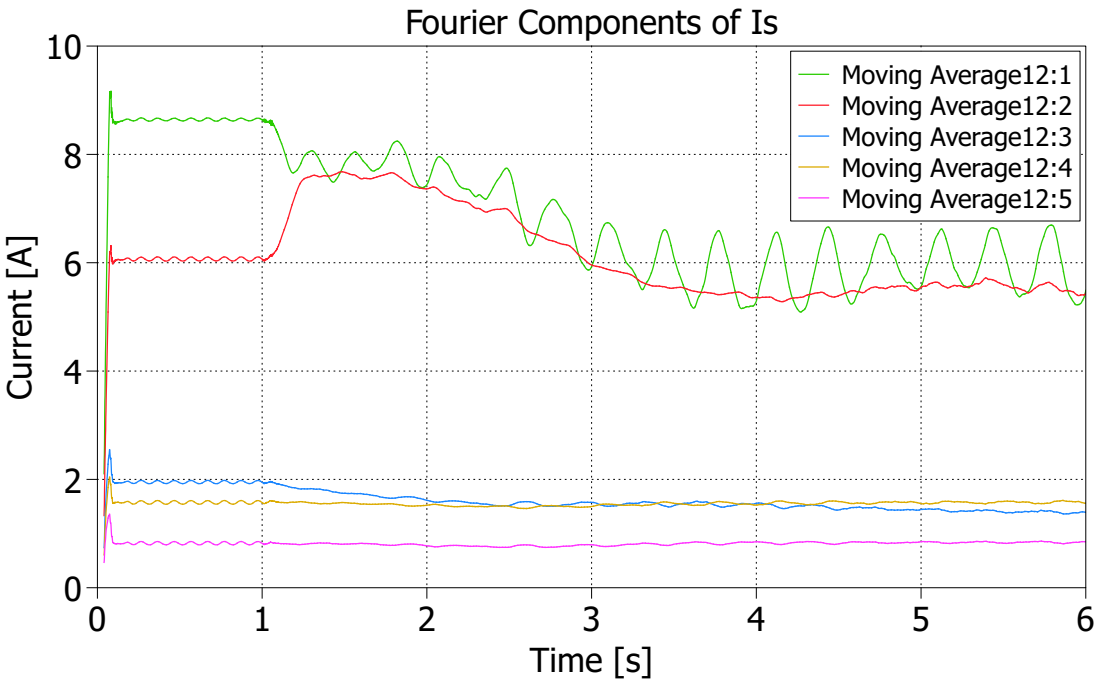


Figure 4.13: The graph of the harmonic amplitudes over time. The LMS algorithm is turned on when one second of simulation time has elapsed. Moving Average12:1 corresponds to the fifth harmonic of the stator currents, Moving Average12:2 to the seventh, Moving Average12:3 to the eleventh and so on.

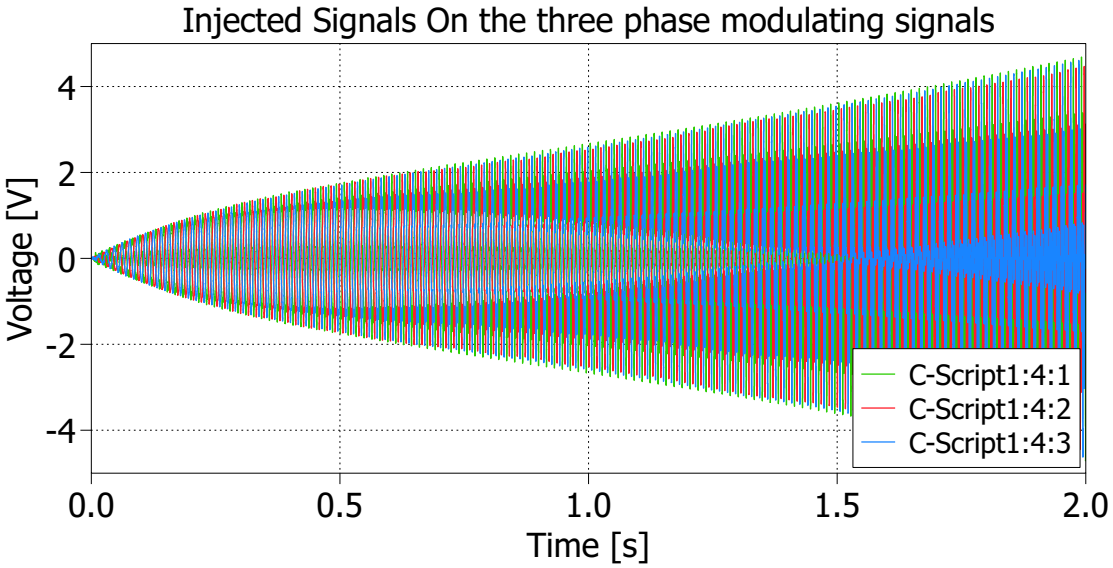


Figure 4.14: Injection signals referring to the LMS-based SHE method, however, the LMS algorithm is overcome by the MPC, as the injection does not stop increasing.

Chapter 5

Conclusions and Recommendations

In this report various techniques for controlling motor drives, were discussed. Main contenders used in the field are mainly FOC and DTC, as discussed in section 2. However, both methods have some drawbacks. For the incorporated FOC method in the simulation, the drawback is the bandwidth of the PI controllers. Accurate setpoint matching can be achieved, however it takes some time to get there. This is where the indirect MPC approach demonstrated its ability to handle the control objectives and system constraints, providing significantly quicker convergence to the references compared to the PI-based approach. Nonetheless, the indirect MPC method struggled to reach the exact current target, as the MPC controller does not have a model incorporated of the plant of the SPWM block. The SPWM modulator that is attached to the MPC controller provides a PWM signal, which can not match the MPC commands instantly.

As for modulation techniques, mainly SPWM modulators are used in the implementation of FOC. Implementing THI or other zero-sequence injections results in the most optimal usage of the available bus, when an inverter is used like in this report. Another modulator choice is the Equal Area Modulator explored in this report. However, this approach did not perform as expected, potentially due to inaccuracies in modulation timing or harmonic content prediction. Fortunately, the Adaptive Selective Harmonic Elimination (ASHE) method, built upon the Least Mean Squares (LMS) algorithm, delivered satisfactory results. The indirect MPC method was able to reduce the harmonics to the lowest level. The ASHE method effectively minimized harmonics and still provided consistent control over the inverter output, demonstrating its robustness and adaptability in dynamic operating conditions.

The required background information from section 2 was successfully used to build a simulation where unwanted harmonics are eliminated. The indirect MPC scheme was especially chosen for the potential to work with SHE techniques and as a direct replacement of the PI controllers. With the output of the MPC being α and β voltages, it is clear that this MPC can pair nicely with the SHE methods.

Overall, the combination of indirect MPC with ASHE shows potential for achieving precise control in multilevel inverters, but further enhancements, such as compensators, filters, plant modeling or advanced observers, may be required for optimal performance.

For further development of harmonic reduction and THD minimization, combining the

MPC and LMS algorithm would be an interesting proposition. The MPC algorithm proved to be very dynamic in terms of reaching the setpoints in a timely fashion, but it also showed superior performance to the LMS-based SHE method incorporated with the PI-based FOC controller. Thus, a combination of the LMS-based SHE method and MPC would be very interesting. Another way of trying to achieve this could be to change the ASHE method a bit. Instead of superpositioning the correction signals of the LMS algorithm on the modulating waveform in the ABC frame, the LMS algorithm could superposition its output onto the i_d and i_q references, by either incorporating Clarke and Park transformations in the existing implementation or by letting the LMS algorithm work in the $\alpha\beta$ or dq frame. The fast nature of the MPC can then be utilized to compensate for the still present harmonics already.

Additionally, the SHE method requires further improvement. As said before, the differences in ”synchronization” of the gating signals may have an effect on the stability of the system. Improving the algorithm of the equal area modulator to generate similar waveforms to the SPWM modulator could yield improvements. An alternate way to look at the issue that presented itself is on the reference side. If the reference takes into account the ”plant” or behavior of the SHE, by first adjusting the present/calculated switching angle list to the reference amplitude, then calculating the current response of that via the model of the motor, the output of the PI would possibly become less noisy, which would help the functioning of the scheme.

Incorporating the plant of the LMS method could also be beneficial in the PI-based approach, as higher bandwidths could be explored. The PI does not compensate for the changes that the LMS algorithm is trying to make in that case.

For the case of the MPC, which did not exactly reach the current references, recommendations are also thought of. The limitation of not reaching the references precisely suggests that the inclusion of an observer or an integrating compensator may be necessary to improve accuracy and address model uncertainties. Papers such as [17] could prove helpful in solving this challenge. Another possible option would be to incorporate a fast-updating FCS-MPC method, to achieve a reasonable performance to PWM. FCS-MPC incorporates functionality similar to a modulator, which means that it can match the references more accurately, because there is no SPWM block in-between. If fast enough updates are not available, other MPC techniques have been researched in section 2.5 (M2PC or OSS-MPC for example) that do incorporate a modulator-like system. Attaching the LMS control voltages to the references of these MPC schemes could prove beneficial.

Bibliography

[1] I. Harbi, J. Rodriguez, E. Liegmann, H. Makhamreh, M. L. Heldwein, M. Novak, M. Rossi, M. Abdelrahem, M. Trabelsi, M. Ahmed, P. Karamanakos, S. Xu, T. Dragičević, and R. Kennel, "Model-predictive control of multilevel inverters: Challenges, recent advances, and trends," *IEEE Transactions on Power Electronics*, vol. 38, no. 9, pp. 10 845–10 868, 2023.

[2] "Vibration damper 800-003 (45-82 kg) - ab marine service." [Online]. Available: https://ab-marineservice.com/en/product/vibration-damper-800-003-45-82-kg/

[3] "Waterworld set of vibration dampers - robust-mt marine technology bv." [Online]. Available: https://www.robust-mt.nl/en/product/waterworld-set-of-vibration-dampers/

[4] "Rubber motor vibration mounts — shock isolators — electric — dampers." [Online]. Available: https://www.avproductsinc.com/cone-shear/motor-mounts.html

[5] NXP, "How pmsm works with field oriented control - nxp community." [Online]. Available: https://community.nxp.com/t5/NXP-Model-Based-Design-Tools/How-PMSM-Works-With-Field-Oriented-Control/ta-p/1129974

[6] S. Raj, R. Aziz, and M. Ahmad, "Influence of pole number on the characteristics of permanent magnet synchronous motor (pmsm)," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 13, pp. 1318–1323, 03 2019.

[7] I. The MathWorks, "Field-oriented control (foc) - matlab simulink - mathworks benelux." [Online]. Available: https://nl.mathworks.com/help/mcb/gs/implement-motor-speed-control-by-using-field-oriented-control-foc.html

[8] Y. Solbakken, "Space vector pwm intro — switchcraft." [Online]. Available: https://www.switchcraft.org/learning/2017/3/15/space-vector-pwm-intro

[9] I. The MathWorks, "Direct torque control (dtc) - matlab simulink - mathworks benelux." [Online]. Available: https://nl.mathworks.com/help/mcb/gs/direct-torque-control-dtc.html

[10] M. A. W. Begh and H.-G. Herzog, "Comparison of field oriented control and direct torque control," *Technical University of Munich, Germany*, 2018.

[11] R. Hutchins, "Efficiency/power factor/harmonics." [Online]. Available: https://web.lamarchemfg.com/userfiles/technical-papers/Charger's%20Efficiency-Power%20Factor%20-%20THD.pdf

[12] "How total harmonic distortion (thd) impacts power system efficiency." [Online]. Available: https://www.alliedcomponents.com/blog/how-total-harmonic-distortion-thd-impacts-power-system-efficiency

[13] D. Ahmadi, K. Zou, C. Li, Y. Huang, and J. Wang, "A universal selective harmonic elimination method for high-power inverters," *IEEE Transactions on Power Electronics*, vol. 26, no. 10, pp. 2743–2752, 2011.

[14] V. Blasko, "A novel method for selective harmonic elimination in power electronic equipment," *IEEE Transactions on Power Electronics*, vol. 22, no. 1, pp. 223–228, 2007.

[15] J. Hong and R. Cao, "Adaptive selective harmonic elimination model predictive control for three-level t-type inverter," *IEEE Access*, vol. 8, pp. 157 983–157 994, 2020.

[16] J. Xu, T. B. Soeiro, F. Gao, L. Chen, H. Tang, P. Bauer, and T. Dragičević, "Carrier-based modulated model predictive control strategy for three-phase two-level vsis," *IEEE Transactions on Energy Conversion*, vol. 36, no. 3, pp. 1673–1687, 2021.

[17] S. Niu, Y. Luo, W. Fu, and X. Zhang, "Robust model predictive control for a three-phase pmsm motor with improved control precision," *IEEE Transactions on Industrial Electronics*, vol. 68, no. 1, pp. 838–849, 2021.

[18] J. Zou, "Deadbeat control combined with finite control set model predictive control for permanent magnet synchronous machine based two-level inverter," in *2021 33rd Chinese Control and Decision Conference (CCDC)*, 2021, pp. 893–898.

[19] plexim, "Plecs: Current controller design for motor drives — plexim." [Online]. Available: https://www.plexim.com/content/current-controller-design-motor-drives

[20] "Iir filter - implement infinite impulse response (iir) filter - simulink - mathworks benelux." [Online]. Available: https://nl.mathworks.com/help/mcb/ref/iirfilter.html

[21] Q. Xuefei, S. Jianxin, N. Robert, and G. Jacek, "Design of high-speed pmsm considering multi-physics fields and power converter constraints," *Transactions of China Electrotechnical Society*, vol. 37, no. 7, pp. 1618–1633, 2022.

[22] D. Lee, T. Balachandran, S. Sirimanna, N. Salk, A. Yoon, P. Xiao, J. Macks, Y. Yu, S. Lin, J. Schuh, P. Powell, and K. S. Haran, "Detailed design and prototyping of a high power density slotless pmsm," *IEEE Transactions on Industry Applications*, vol. 59, no. 2, pp. 1719–1727, 2023.

Appendix A

Additional Figures

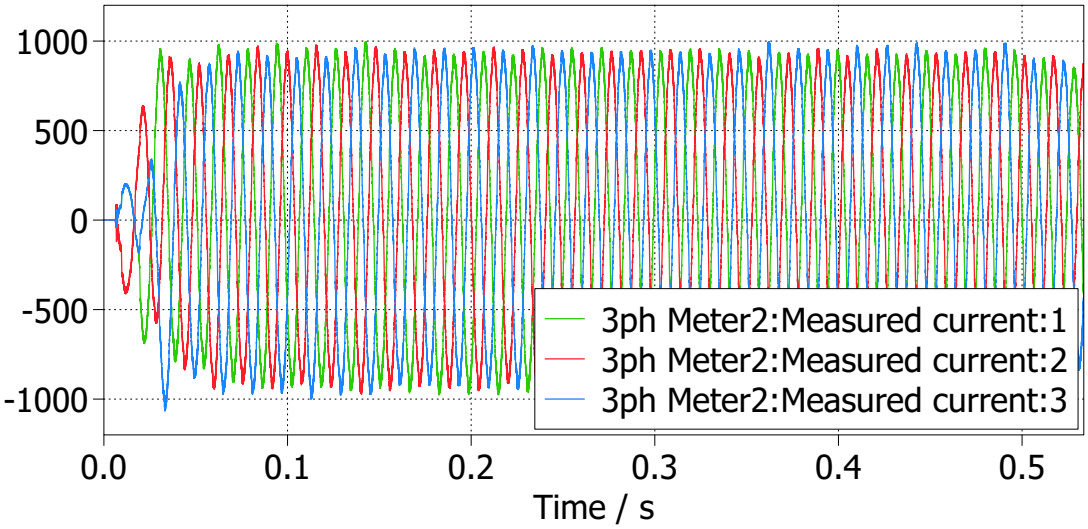


Figure A.1: The modulator output of the SHE algorithm was passed to an RL circuit. The resulting phase currents are displayed in this figure.

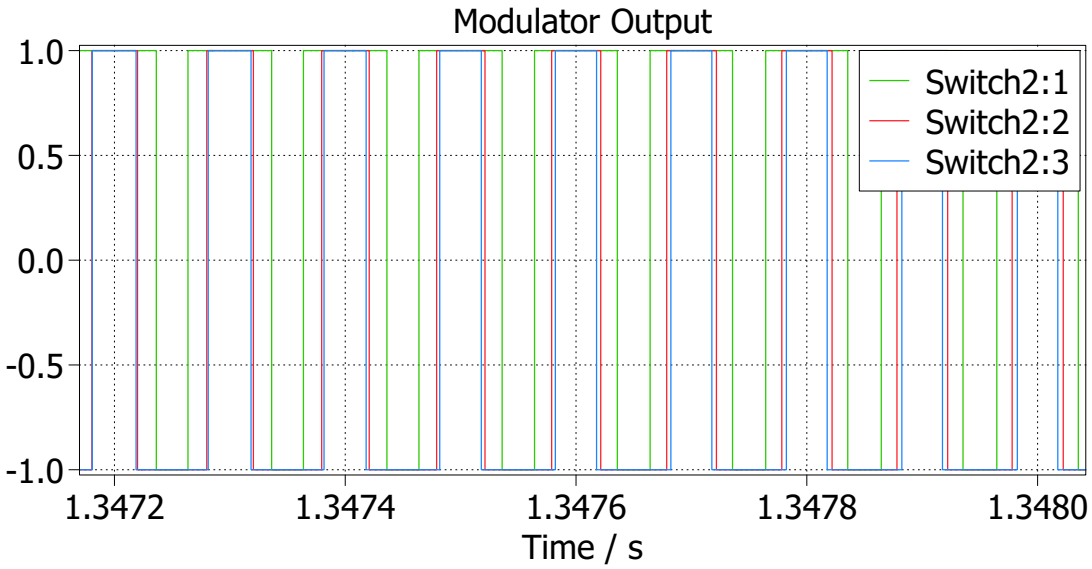


Figure A.2: Modulator output with no SHE active

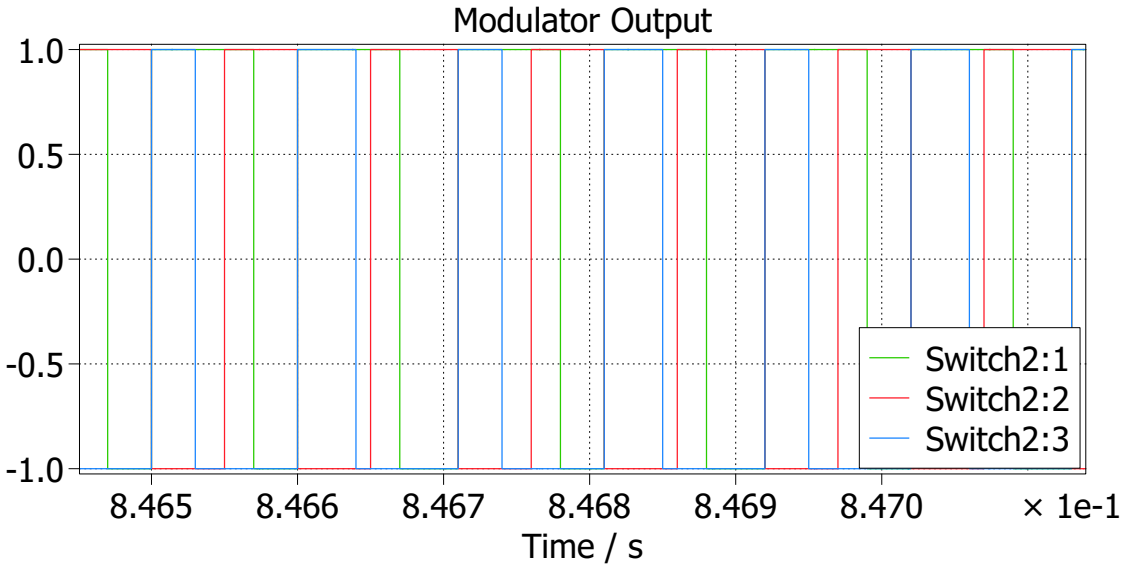


Figure A.3: Modulator output with SHE active

Appendix B

Third Harmonic Injection Derivation

To recap what was written in section 2.4:

A start is made by writing down the equation for the modulating signal with the superpositioned third harmonic.

$$m(x) = sin(x) + asin(3x), \mathbb{D} : \{x | x \in [0, 2\pi]\} \tag{B.1}$$

The derivative is computed with respect to x.

$$m'(x) = cos(x) + 3acos(3x) \tag{B.2}$$

The minima and maxima are of interest:

$$m'(x) = 0 \tag{B.3}$$

To solve the equation

$$\cos(x) + 3a\cos(3x) = 0 \tag{B.4}$$

for x, the derivation is provided here:

Step 1: Express $\cos(3x)$ in terms of $\cos(x)$

Using the triple-angle formula:

$$\cos(3x) = 4\cos^3(x) - 3\cos(x) \tag{B.5}$$

The triple-angle formula can itself be derived with the following trigonometric identity:

$$\cos(A + B) = \cos(A)\cos(B) - \sin(A)\sin(B)$$

Step 2: Substitute equation B.5 in the original equation

$$\cos(x) + 3a(4\cos^3(x) - 3\cos(x)) = 0 \tag{B.6}$$

$$\cos(x) + 12a\cos^3(x) - 9a\cos(x) = 0 \tag{B.7}$$

$$\cos(x)(1 - 9a + 12a\cos^2(x)) = 0 \tag{B.8}$$

Then $\cos(x) = 0$, $x = \frac{\pi}{2} + k\pi, \quad k \in \mathbb{Z}$ or $1 - 9a + 12a\cos^2(x) = 0$

The derived values for x in the first solution do not include a in the equation, so the second solution needs to be found by solving for $\cos^2(x)$:

$$12a\cos^2(x) = 9a - 1 \tag{B.9}$$

$$\cos^2(x) = \frac{9a-1}{12a} \tag{B.10}$$

Taking the square root:

$$\cos(x) = \pm\sqrt{\frac{9a-1}{12a}} \tag{B.11}$$

Therefore, the solutions are:

$$x = \pm cos^{-1}\left(\sqrt{\frac{9a-1}{12a}}\right) + 2k\pi, \quad k \in \mathbb{Z} \tag{B.12}$$

Substituting the solution from equation B.12 into the starting equation (B.1):

$$m(cos^{-1}\left(\sqrt{\frac{9a-1}{12a}}\right)) = sin(cos^{-1}\left(\sqrt{\frac{9a-1}{12a}}\right)) + asin(3cos^{-1}\left(\sqrt{\frac{9a-1}{12a}}\right)) \tag{B.13}$$

The last term of equation B.13 can be taken as $\sin(3\cos^{-1}(x))$, this term can then be simplified:

1. Let $\theta = \cos^{-1}(x)$, hence $\cos(\theta) = x$: Since $\cos(\theta) = x$, we can use trigonometric identities to express $\sin(3\theta)$ in terms of $\cos(\theta)$.

2. Use the triple angle formula for sine:

$$\sin(3\theta) = 3\sin(\theta) - 4\sin^3(\theta)$$

To find $\sin(\theta)$, recall that $\sin^2(\theta) + \cos^2(\theta) = 1$. Therefore,

$$\sin(\theta) = \sqrt{1 - \cos^2(\theta)} = \sqrt{1 - x^2}$$

3. Substitute $\sin(\theta)$ and $\cos(\theta)$ into the triple angle formula:

$$\sin(3\cos^{-1}(x)) = 3\sin(\cos^{-1}(x)) - 4\sin^3(\cos^{-1}(x))$$

$$\sin(\cos^{-1}(x)) = \sqrt{1 - x^2} \tag{B.14}$$

Therefore,

$$\sin(3\cos^{-1}(x)) = 3\sqrt{1 - x^2} - 4(\sqrt{1 - x^2})^3$$

4. Simplify the expression:

$$\sin(3\cos^{-1}(x)) = 3\sqrt{1 - x^2} - 4(1 - x^2)^{3/2}$$

$$= 3\sqrt{1 - x^2} - 4(1 - x^2)\sqrt{1 - x^2}$$

$$= 3\sqrt{1 - x^2} - 4(1 - x^2)^{3/2}$$

Thus:

$$\sin(3\cos^{-1}(x)) = 3\sqrt{1-x^2} - 4(1-x^2)^{3/2} \tag{B.15}$$

Accordingly, equation B.14 can be used to simplify the first term of equation B.13 and equation B.15 is used to simplify the second term of equation B.13:

$$m(cos^{-1}\left(\sqrt{\frac{9a-1}{12a}}\right)) = \sqrt{1-\frac{9a-1}{12a}} + a\left(3\cdot\sqrt{1-\frac{9a-1}{12a}} - 4\left(1-\frac{9a-1}{12a}\right)^{\frac{3}{2}}\right) \tag{B.16}$$

Equation B.16 can then be differentiated with respect to a, with the goal to obtain a value of a connected to an extreme point in the output of equation B.16, which corresponds to the ideal amplitude of the third harmonic to be injected. The derivation can be performed by using the differentiation rules $(f \pm g)' = f' \pm g'$ and $(fg)' = f' \cdot g + f \cdot g'$.

$$\begin{aligned}\frac{d}{da}\left(\sqrt{1-\frac{9a-1}{12a}} + a\left(3\cdot\sqrt{1-\frac{9a-1}{12a}} - 4\left(1-\frac{9a-1}{12a}\right)^{\frac{3}{2}}\right)\right) = \\ -\frac{1}{4\cdot\ 3^{\frac{1}{2}}x^{\frac{3}{2}}\left(3x+1\right)^{\frac{1}{2}}} + \frac{36x^2+6x+1}{12\cdot\ 3^{\frac{1}{2}}x^{\frac{3}{2}}\left(3x+1\right)^{\frac{1}{2}}} = \\ \frac{\left(6x-1\right)\left(3x+1\right)^{\frac{1}{2}}}{6\cdot\ 3^{\frac{1}{2}}x^{\frac{3}{2}}}\end{aligned} \tag{B.17}$$

Setting the final solution of equation B.17 equal to zero yields the optimal amplitude for THI:

$$\frac{\left(6x-1\right)\left(3x+1\right)^{0.5}}{6\cdot 3^{0.5}\cdot\ x^{\frac{3}{2}}} = 0 \tag{B.18}$$

$$x = \frac{1}{6} \tag{B.19}$$

By using the found amplitude from equation B.19 in equation B.21, the gained amount of bus utilisation can be calculated. The value used for x can be calculated using equation B.12.

$$x = \cos^{-1}\left(\sqrt{\frac{\frac{9}{6}-1}{\frac{12}{6}}}\right) = \frac{\pi}{3} \tag{B.20}$$

$$m(\frac{\pi}{3}) = sin(\frac{\pi}{3}) + \frac{1}{6}sin(3\cdot\frac{\pi}{3}) = \frac{\sqrt{3}}{2} \tag{B.21}$$

Thus the gained bus utilisation is: $1/\frac{\sqrt{3}}{2} = \frac{2}{\sqrt{3}} = 1.1547$ or 115.47% bus utilisation.

Appendix C

Code of the PLL

C.0.1 Definitions and Functions of the code

```
#define pIn_time   0
#define pIn_angle   1
#define pIn_VdqGoal  2
#define pIn_mSpeed  3
#define pIn_SWALPHAS  4
#define pIn_MPP   5

#define pOut_phaseABC 0
#define pOut_ABCfilt  1
#define pOut_phaseOut 2
#define pOut_cutFreq  3

#define N_PHASES    4
#define SAMPLE_FREQ    100000

#define FILT      1
#define FILT_MIN_CUT_FREQ 1
#define FILT_ORDER   6 //PI adds another order
#define FILT_FREQ_GOAL  2500
#define FILT_ATT_GOAL  -40 //dB
#define FILT_DYN_TARGET  0

#define DYN_MODE    2 //1 = freq goal @influence due to SWALPHAS, 2 = freq
    ↪ goal @influence due to motorspeed itself

#define INIT_SPEED   100
#define PI_DYN_TARGET  1
#define PI_FREQ_GOAL   40
```

```
#define PI_ATT_GOAL   -0
#define PI_MIN_CUT_FREQ  1
#define PI_FILT_PHASE_TOT -6 //this much phase margin is added from the (
    ↪ cascaded) filter at fcut from PI. -114 -> -180 ; -66

#define PLANT_K0    1

//#define out_phaseA OutputSignal(pOut_phaseABC, 0)
//#define out_phaseB OutputSignal(pOut_phaseABC, 1)
//#define out_phaseC OutputSignal(pOut_phaseABC, 2)

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

float time = 0, correction = 0, phase[N_PHASES] = {0}, phaseModded[N_PHASES]
    ↪  = {0}, Vd = 0, Vq = 0;
int swAlphas, MPP;
//double filtCutFreq = 0, decades = 0;

typedef struct motor {
       float motAngle, motAngleLast, speed;
} motor;
motor m = {0};

typedef struct lpf {
 double cutFreq, decades;
 float a, yLast[FILT_ORDER][N_PHASES], y[FILT_ORDER][N_PHASES], out[N_PHASES
     ↪ ], in[N_PHASES];

} lpf;
lpf filt = {0};

typedef struct PI {
       float Kp, Ki, I[N_PHASES], P[N_PHASES], e[N_PHASES], out[N_PHASES],
           ↪ tlast[N_PHASES];
       double cutFreq, decades;
       //int FcutDivisor;
} PI;

PI C = {0};
```

```
typedef struct plant {
        int K0;
        float out[N_PHASES], outModded[N_PHASES], tlast[N_PHASES];

} plant;

plant P = {0};


double signD(double num) {
        if(num < 0) {
                return -1;
        } else {
                return 1;
        }
}

int signI(int num) {
        if(num < 0) {
                return -1;
        } else {
                return 1;
        }
}

float signF(float num) {
        if(num < 0) {
                return -1;
        } else {
                return 1;
        }
}
/**
 * Find maximum between two numbers.
 */
float max(float num1, float num2)
{
    return (num1 > num2 ) ? num1 : num2;
}

/**
 * Find minimum between two numbers.
```

```
 */
float min(float num1, float num2)
{
    return (num1 > num2 ) ? num2 : num1;
}

float constrain(float num, float lo, float hi) {
        return max(min(num, hi), lo);
}

float avg(float num1, float num2) {

        return (num1+num2)/2;

}
```

C.0.2 Start of the code

```
P.K0 = PLANT_K0;

if(!FILT_DYN_TARGET) {

        filt.decades = (float)FILT_ATT_GOAL/((float)FILT*(float)FILT_ORDER
            ↪ *-20);
        filt.cutFreq = (float)FILT_FREQ_GOAL/pow(10, filt.decades);
        filt.a = 2*M_PI*(1/(float)SAMPLE_FREQ)*filt.cutFreq/(2*M_PI*(1/(float
            ↪ )SAMPLE_FREQ)*filt.cutFreq+1);
}

if(!PI_DYN_TARGET) {

        C.decades = (float)PI_ATT_GOAL/((float)-20);
        C.cutFreq = (float)PI_FREQ_GOAL/pow(10, C.decades);
} else {

        double pm = -45-(float)PI_FILT_PHASE_TOT/FILT_ORDER;
        C.decades = pm/-45;
        C.cutFreq = filt.cutFreq/pow(10, C.decades);
}
C.Kp = C.cutFreq*2*M_PI*sqrt(2*sqrt(2)-2)/P.K0;
C.Ki = pow((C.cutFreq*2*M_PI), 2)*(sqrt(2)-1)/P.K0;
```

```
MPP = InputSignal(pIn_MPP, 0);

for(int i = 0; i < N_PHASES; i++) {
       C.I[i] = INIT_SPEED;
}
```

C.0.3 Updating part of the code

```
    time = InputSignal(pIn_time, 0);
m.motAngleLast = m.motAngle;
m.motAngle = InputSignal(pIn_angle, 0);
Vd = InputSignal(pIn_VdqGoal, 0);
Vq = InputSignal(pIn_VdqGoal, 1);
m.speed = InputSignal(pIn_mSpeed, 0);

swAlphas = InputSignal(pIn_SWALPHAS, 0);

OutputSignal(pOut_cutFreq, 0) = filt.cutFreq;
OutputSignal(pOut_cutFreq, 1) = C.cutFreq;

if(FILT_DYN_TARGET) {

       filt.decades = (float)FILT_ATT_GOAL/((float)FILT*(float)FILT_ORDER
           ↪ *-20);
       //filtCutFreq = FILTPI_FREQ_GOAL/pow(10, decades);
       if(DYN_MODE == 1) {
              filt.cutFreq = max(m.speed/(2*M_PI)*MPP*swAlphas/2*3/pow(10,
                  ↪ filt.decades),FILT_MIN_CUT_FREQ);
       } else if(DYN_MODE == 2) {
              filt.cutFreq = max(m.speed/(2*M_PI)*MPP*3/pow(10, filt.decades
                  ↪ ),FILT_MIN_CUT_FREQ);
       }
       filt.a = 2*M_PI*(1/(float)SAMPLE_FREQ)*filt.cutFreq/(2*M_PI*(1/(float
           ↪ )SAMPLE_FREQ)*filt.cutFreq+1);

       if(PI_DYN_TARGET) {
              double pm = -45-(float)PI_FILT_PHASE_TOT/FILT_ORDER;
              C.decades = pm/-45;
              C.cutFreq = filt.cutFreq/pow(10, C.decades);
              C.Kp = C.cutFreq*2*M_PI*sqrt(2*sqrt(2)-2)/P.K0;
```

```
                C.Ki = pow((C.cutFreq*2*M_PI), 2)*(sqrt(2)-1)/P.K0;
        }

}

/*if(Vd > 0 && Vq >= 0) correction = atanf(Vq/Vd);
else if(Vd < 0) correction = atanf(Vq/Vd)+M_PI;
else if(Vd > 0 && Vq < 0) correction = atanf(Vq/Vd+2*M_PI);
else if(Vd == 0) correction = signF(Vq)*0.5*M_PI;*/

correction = atan2f(Vq, Vd);

for(int i = 0; i < N_PHASES; i++) {
        //Phase measured from motor angle

        if(i != 3) {
                phase[i] = m.motAngle+correction-(float)i/3*2*M_PI+0.5*M_PI;
        } else {
                phase[i] = m.motAngle;
        }

        if(signF(m.motAngle) == -1 && signF(m.motAngleLast) == 1 && m.speed >
            ↪  0) P.out[i] = phase[i]-C.e[i];
        else if(signF(m.motAngle) == 1 && signF(m.motAngleLast) == -1 && m.
            ↪ speed < 0) P.out[i] = phase[i]+C.e[i];

        phaseModded[i] = fmodf(phase[i], 2*M_PI);
        if(phaseModded[i] < 0) phaseModded[i] += 2*M_PI;

        OutputSignal(pOut_phaseABC, i) = phaseModded[i];
        if(FILT) {
        filt.in[i] = phase[i]-P.out[i]; //error calculation into filter
        //if(filt.in[i] > M_PI) filt.in[i] -= 2*M_PI;
        //else if(filt.in[i] < -M_PI) filt.in[i] += 2*M_PI;


        for(int j = 0; j < FILT_ORDER; j++) {
                if(j==0) filt.y[j][i] = (filt.in[i])*filt.a + (1-filt.a)*filt.
                    ↪ yLast[j][i];
                else  filt.y[j][i] = filt.y[j-1][i]*filt.a + (1-filt.a)*filt.
                    ↪ yLast[j][i];
```

```
        filt.yLast[j][i] = filt.y[j][i];
    }
    filt.out[i] = filt.y[FILT_ORDER-1][i];



    }
    //Controller
    if(FILT) {
        C.e[i] = filt.out[i]; //error into PI
    } else {
        /*typedef struct PI {
        float Kp, Ki, I[3], P[3], e[3], out[3];
        double cutFreq;
        } PI;*/
        C.e[i] = phase[i]-P.out[i]; //error into PI
        //if(filt.in[i] > M_PI) filt.in[i] -= 2*M_PI;
        //else if(filt.in[i] < -M_PI) filt.in[i] += 2*M_PI;
    }

    //float temp = phase[i]-P.out[i]; //error into PI
        //if(temp > M_PI) temp -= 2*M_PI;
        //else if(temp < -M_PI) temp += 2*M_PI;
    OutputSignal(pOut_ABCfilt, i) = C.e[i];

    C.P[i] = C.Kp*C.e[i];
    C.I[i] = C.Ki*C.e[i]*(time-C.tlast[i])+C.I[i];
    C.tlast[i] = time;
    C.out[i] = C.P[i] + C.I[i];

    //Plant
    P.out[i] = C.out[i]*(time-P.tlast[i])+P.out[i];
    P.tlast[i] = time;
    P.outModded[i] = fmodf(P.out[i], 2*M_PI);
    if(P.outModded[i] < 0) P.outModded[i] += 2*M_PI;

    OutputSignal(pOut_phaseOut, i) = P.outModded[i];

    //int nPhase = phase[i]/(2*M_PI), nPout = P.out[i]/(2*M_PI), nFin;
    /*if(signI(nPhase) == 1 && signI(nPout) == 1) {
        nFin = min(nPhase, nPout);
        phase[i] -= nFin*2*M_PI;
```

```
                P.out[i] -= nFin*2*M_PI;
        }else if(signI(nPhase) == -1 && signI(nPout) == -1) {
                nFin = max(nPhase, nPout);
                phase[i] -= nFin*2*M_PI;
                P.out[i] -= nFin*2*M_PI;
        }*/
}

//printf("@: %f, a: %f, cutFreq: %f, Va phase unfiltered: %f, C.cutFreq: %f,
    ↪  C.e: %f, C.P: %f, C.I: %f, C.out: %f, P.out: %f\n", time, filt.a,
    ↪ filt.cutFreq, phase[0], C.cutFreq, C.e[0], C.P[0], C.I[0], C.out[0], P
    ↪ .out[0]);
```

Appendix D

Code of the SHE algorithm

D.0.1 Definitions and Functions of the code

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define N_PHASES 3
//#define M_PI acos(-1.0)
#define PWMTOL 0.75
//#define SWALPHAS 40 //NEEDS to be multiple of 4 per whole period

#define CALCMODE 1; //1 is full periods, 2 is half periods

int HCs[] = {1, 3, 5, 7, 9, 11, 13, 35, 37, 41, 43, 73, 77, 79, 83};
//int HCsSHE[] = {5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47,
    ↪ 49, 53, 55, 59, 61, 65, 67, 71, 73, 77, 79, 83};

int HCsSHE[] = {5, 7};
//int HCsSHE[] = {5, 7, 11, 13, 17};

#define SHE 1 //set SHE on or off
#define SHE_ITERATIONS  100 //max iterations
#define SHE_ELIMINATED  0.1 // at this value, harmonics are considered
    ↪ eliminated
#define SHE_FUNDAMENTAL  1
#define SHE_3H_INJECTION 0
#define THETAS_SWITCHING_WITH_SIGMAS 1

#define VF_SAMPLES_PP 10 //number of samples per period
```

```
#define VF_OVER_X_PERIODS 1 //array is filled in x phase periods
#define VF_DELAY_PERIODS 0
#define VF_MIN_SAMPLE_T 0.02
#define VF_LIVE 1
#define VF_SAMPLES_TOT VF_SAMPLES_PP*VF_OVER_X_PERIODS

#define FIRSTMOVE_ALTERING 0
//#define FIRSTMOVE_MODE 0 //0 = every period first move alters; 1 = every
    ↪ first half is flipped mode; 2 = every second half is flipped mode; 3 =
    ↪  1 and 2 alternated every period; 4 = random every half period

#define randnum(min, max) \
       ((rand() % (int)(((max) + 1) - (min))) + (min))

#define pIn_Vpn    0
#define pIn_VdqGoal   7
#define pIn_VnGoal   8
#define pIn_swAlphas   9
#define pIn_FPIPLL   10

#define pOut_HCsVa   0
#define pOut_Mod    3
#define pOut_PeriodUpdate 4
#define pOut_Phase   6
#define pOut_Speed   7
#define pOut_SWALPHAS  8

#define inVdc     InputSignal(1,0)
#define inThreshold   InputSignal(2,0)
#define inTime     InputSignal(3,0)
#define inRotorSpeed   InputSignal(4,0)
#define inRotorAngle   InputSignal(4,1)
#define inPolePairs   InputSignal(5,0)
#define inFpwm     InputSignal(6,0)

#define outTime    OutputSignal(5,0)

double Vdc = 0, VdcHalf = 0, lastVPN1State = 0, lastVPN2State = 0,
    ↪ lastVPN3State = 0, lastVPN1 = 0, lastVPN2 = 0, lastVPN3 = 0;

int SHE_arrSize = sizeof(HCsSHE)/sizeof(HCsSHE[0])+SHE_FUNDAMENTAL +
    ↪ SHE_3H_INJECTION;
```

```
float dTheta, VdGoalLast, VqGoalLast, VdGoal, VqGoal; //phase between two
    ↪ sigma k's

int threshold = 0, calcMode = CALCMODE;
int sampledPhases = 0, sheUpdatedPhases = 0;
int swAlphas = 0; // Harmonics to calculate


typedef struct electricMotor {
        double rotorSpeed, rotorAngle;
        int polePairs;

} electricMotor;

float time = 0;
float slowPrinter = 0, maxVoltageTimer = 0, sheUpdateTime = 0;

// Structure to hold switching timestamps
typedef struct switchingTimestamp {
    float timestamp;
    double voltage;
    //float
    struct switchingTimestamp *next;
    struct switchingTimestamp *prev;

} switchingTimestamp;

struct switchingTimestamp *switchingTimestampsVPN[N_PHASES] = {NULL};
struct switchingTimestamp *comingStampsFullV[N_PHASES] = {NULL};
struct switchingTimestamp *comingStampsQuartV[N_PHASES] = {NULL};
struct switchingTimestamp *comingStampsQuartBeginV[N_PHASES] = {NULL};

typedef struct phaseStruct {
    double voltage, prevVoltage;
    float beginTimePeriod, deltaT, periodAverage, lastPeriodAverage,
        ↪ lastRiseTimestamp,
    lastFallTimestamp, lastHalfPeriodTimestamp, lastHalfPeriodTime,
        ↪ maxVoltage, goalVoltage,
    prevGoalVoltage, lastPeriodTimestamp, phase, lastPhase, rotorSpeed,
        ↪ thetaMid, nextVSampleTimestamp;
```

```
    int state, listSize, halfBridgeState, dThetasPassed, dThetasPassedLast,
        ↪ samplesIndex;
    bool modReset, modState, stateReset, stateReset2, comingNegativePeriod,
        ↪ triggerRefresh, firstMoveState;
    float comingHarmonics[sizeof(HCsSHE)/sizeof(HCsSHE[0])+SHE_FUNDAMENTAL +
        ↪ SHE_3H_INJECTION], VSamples[VF_SAMPLES_TOT];
    switchingTimestamp *alphaTraverser;
} phaseStruct;


typedef struct pulsewidthmodulation {
        double F, tolerance;
        float T, startTime, onTime, endTime;
} pulsewidthmodulation;

struct electricMotor motor = {0};

//struct phaseStruct phaseEMPTY = {0};
struct phaseStruct phase[N_PHASES] = {0};// = {NULL, NULL, NULL};

struct pulsewidthmodulation pwm = {0};


//float fmod(float num, float divisor) {
// return num-((int)(num/divisor))*divisor;
//}

double signD(double num) {
        if(num < 0) {
                return -1;
        } else {
                return 1;
        }
}

int signI(int num) {
        if(num < 0) {
                return -1;
        } else {
                return 1;
        }
}
```

```
float signF(float num) {
       if(num < 0) {
              return -1;
       } else {
              return 1;
       }
}
/**
 * Find maximum between two numbers.
 */
float max(float num1, float num2)
{
    return (num1 > num2 ) ? num1 : num2;
}

/**
 * Find minimum between two numbers.
 */
float min(float num1, float num2)
{
    return (num1 > num2 ) ? num2 : num1;
}

float constrain(float num, float lo, float hi) {
       return max(min(num, hi), lo);
}

float avg(float num1, float num2) {

       return (num1+num2)/2;

}

float averageMotorSpeeds(float samples[], int* index, float rotorSpeed,
    ↪ float* outputTimestamp) {


       samples[VF_SAMPLES_TOT-1] = rotorSpeed;
       if(index != NULL) *index = 0;
  int divisor = VF_SAMPLES_TOT;
  float temp = 0;
```

```
	for(int i = 0; i < VF_SAMPLES_TOT; i++) {
		if(fabs(samples[i]) > 0) {
			temp += samples[i];
		} else {
			divisor--;
		}
		//printf("Sample %d: %f, ", i, samples[i]);
	}
	float outputSpeed = temp/divisor;
	if(outputSpeed != 0 && outputTimestamp != NULL) {
		*outputTimestamp = time+2*M_PI/rotorSpeed/VF_SAMPLES_PP;
	}
	return outputSpeed;
}
float averageMotorSpeedsLive(float samples[], int index) {

	index--;
	if(index<0) index = VF_SAMPLES_TOT-1;
  int divisor = VF_SAMPLES_TOT-VF_DELAY_PERIODS*VF_SAMPLES_PP;
  float temp = 0;
  int iEnd = index - 1 - VF_DELAY_PERIODS*VF_SAMPLES_PP;
  if(iEnd < 0) iEnd += VF_SAMPLES_TOT;

  int i = index;
  bool going = true;
  //int loopCounter = 0;
  while(going) {
		//going = false;
		if(i >= VF_SAMPLES_TOT) i = 0;
		if(fabs(samples[i]) > 0) {
			temp += samples[i];
		} else {
			divisor--;
		}

		if(i == iEnd) going = false;
		//printf("LoopCounter: %d, i: %d, iEnd: %d, divisor: %d, temp:
		    ↪ %f\n", loopCounter, i, iEnd, divisor, temp);
		i++;

		//if(loopCounter > 500) going = false;
		//loopCounter++;
```

```
        }
        return temp/divisor;
}
// Function to check if harmonic elimination is reached
bool completedSHE(float comingHarmonics[], float maxVoltage) {

        int enabledFeatures = 0;
        if(SHE_FUNDAMENTAL) {
                if(fabs(comingHarmonics[enabledFeatures]-maxVoltage) >
                    ↪ SHE_ELIMINATED) return false;
                enabledFeatures++;
        }
        if(SHE_3H_INJECTION) {
                if(fabs(comingHarmonics[enabledFeatures]-maxVoltage/6) >
                    ↪ SHE_ELIMINATED) return false;
                enabledFeatures++;
        }

        for(int j = enabledFeatures; j < SHE_arrSize; j++) {
                if(fabs(comingHarmonics[j]) > SHE_ELIMINATED) return false;
        }

        return true;
}
// Function to add a new switching timestamp to the list
void addswitchingTimestamp(switchingTimestamp **head, float timestamp,
    ↪ double voltage) {
    switchingTimestamp *newTimestamp = (switchingTimestamp *)malloc(sizeof(
        ↪ switchingTimestamp));
    if (newTimestamp == NULL) {
        fprintf(stderr, "Memory␣allocation␣failed\n");
        exit(EXIT_FAILURE);
    }

    newTimestamp->timestamp = timestamp;
    newTimestamp->voltage = voltage;
    newTimestamp->next = *head;
    newTimestamp->prev = NULL;

    if (*head != NULL) {
        (*head)->prev = newTimestamp;
```

```
    }

    *head = newTimestamp;
}

// Function to calculate Fourier expansion for specified harmonics
float calculateHarmonics(double Vdc, switchingTimestamp *head, float *
    ↪ harmonics, float beginTime, int cMode, float in2ndHalf) {

        if(head == NULL) return -1;
    float lastTimestamp = 0; //start at ending = always a half period is
        ↪ completed
    float endTime = head->timestamp;
    float deltaT = endTime - beginTime;
    //deltaT = deltaT*1.1;

    //printf("rotorSpeed: %f, endTime: %f\n", rotorSpeed, endTime);

    for (int m = 0; m < sizeof(HCs)/sizeof(HCs[0]); m++) {
        float sum = 0.0;
        switchingTimestamp *current = head;
        lastTimestamp = current->timestamp;
        //lastTimestamp = timestamp;
        //printf("\n%.4f ", (lastTimestamp-*beginTime)/deltaT*M_PI);
        if(current->next == NULL) return -1;
        current = current->next;
        float timestamp = current->timestamp;

        while (current != NULL) {
            timestamp = current->timestamp;

            //if(m==0) printf("Timestamp: %f, voltage: %f, cMode: %d, sum: %f
                ↪ \n", timestamp, current->voltage, calcMode, sum); //print
                ↪ the full variable size list only one time

            // sum += (current->voltage) / ((double)m * M_PI) * (cos((double)
                ↪ m * timestamp * rotorSpeed * polePairs) - cos((double)m *
                ↪ M_PI)); //rotorspeed = omega of rotor. * polepairs ->
                ↪ electrical rotational speed
            //} else {
            if(cMode == 1) {
```

```
            sum += (current->voltage) / ((float)HCs[m] * M_PI) * (cos((
                ↪ float)HCs[m]* ((timestamp-beginTime)/(deltaT)*2*M_PI))
                ↪ - cos((float)HCs[m]* ((lastTimestamp-beginTime)/(deltaT
                ↪ )*2*M_PI))); //rotorspeed = omega of rotor. * polepairs
                ↪  -> electrical rotational speed

        } else if(cMode == 2) {
            sum += 2*(current->voltage) / ((float)HCs[m] * M_PI) * (cos((
                ↪ float)HCs[m]* (in2ndHalf*M_PI+(timestamp-beginTime)/(
                ↪ deltaT*2)*2*M_PI)) - cos((float)HCs[m]* (in2ndHalf*M_PI
                ↪ +(lastTimestamp-beginTime)/(deltaT*2)*2*M_PI))); //
                ↪ rotorspeed = omega of rotor. * polepairs -> electrical
                ↪ rotational speed

        }


                        //sum += (current->voltage) / ((float)m * M_PI)
                            ↪  * (cos((float)m* (timestamp-beginTime)*
                            ↪ rotorSpeed*motor.polePairs) - cos((float)
                            ↪ m* (lastTimestamp-beginTime)*rotorSpeed*
                            ↪ motor.polePairs)); //rotorspeed = omega
                            ↪ of rotor. * polepairs -> electrical
                            ↪ rotational speed

                        //}
                        //printf("%.4f ", (timestamp-*beginTime)/deltaT
                            ↪ *M_PI);
            lastTimestamp = timestamp;
            current = current->next;
        }
        harmonics[m] = sum;
    }
    return deltaT;
}


// Function to calculate Fourier expansion for specified harmonics
float calculateComingHarmonics(switchingTimestamp *head, float *harmonics) {

        if(head == NULL) return -1;
```

```
float lastTheta = 0; //start at ending = always a half period is
    ↪ completed

//deltaT = deltaT*1.1;

//printf("beginTime: %f, endTime: %f, deltaT: %f\n", beginTime, endTime,
    ↪ deltaT);
//printf("inComingHarmonics: %f, ", head->voltage);
//return 0;
for (int m = 0; m < SHE_arrSize; m++) {
    float sum = 0.0;
    switchingTimestamp *current = head;
    lastTheta = current->timestamp;
    //printf("inComingHarmonics: %f, ", lastTheta);
    //lastTheta = timestamp;
    //printf("\n%.4f ", (lastTheta-*beginTime)/deltaT*M_PI);
    current = current->next;
    if(current == NULL) return -1;
    float theta = current->timestamp;
    //printf("inComingHarmonics2: %f, ", theta);
    while (current != NULL) {

        theta = current->timestamp;
        /*
        if(m==1) { printf("phase: %f, voltage: %f\n", theta, current->
            ↪ voltage); //print the full variable size list only one time
        }
        //*/
        // sum += (current->voltage) / ((double)m * M_PI) * (cos((double)
            ↪ m * timestamp * rotorSpeed * polePairs) - cos((double)m *
            ↪ M_PI)); //rotorspeed = omega of rotor. * polepairs ->
            ↪ electrical rotational speed
        //} else {

        int enabledFeatures = 0;
                        if(SHE_FUNDAMENTAL) {
                                if(m == enabledFeatures) sum += 4*(
                                    ↪ current->voltage) / ((float)1 *
                                    ↪ M_PI) * (cos((float)1 * theta) -
                                    ↪ cos((float)1 * lastTheta)); //
                                    ↪ rotorspeed = omega of rotor. *
                                    ↪ polepairs -> electrical rotational
```

```
                                        ↪  speed
                                enabledFeatures++;
                            }
                            if(SHE_3H_INJECTION) {
                                if(m == enabledFeatures) sum += 4*(
                                    ↪ current->voltage) / ((float)3 *
                                    ↪ M_PI) * (cos((float)3 * theta) -
                                    ↪ cos((float)3 * lastTheta)); //
                                    ↪ rotorspeed = omega of rotor. *
                                    ↪ polepairs -> electrical rotational
                                    ↪  speed
                                enabledFeatures++;
                            }

                            if(m >= enabledFeatures) {
                sum += 4*(current->voltage) / ((float)HCsSHE[m-enabledFeatures
                    ↪ ] * M_PI) * (cos((float)HCsSHE[m-enabledFeatures] *
                    ↪ theta) - cos((float)HCsSHE[m-enabledFeatures] *
                    ↪ lastTheta)); //rotorspeed = omega of rotor. * polepairs
                    ↪  -> electrical rotational speed
                            }
                            //}

                            //printf("%.4f ", (timestamp-*beginTime)/deltaT
                                ↪ *M_PI);
            lastTheta = theta;
            current = current->next;
        }

        harmonics[m] = sum;
    }
    return 1;
}
float calculateThetaK(float thetaLo, float thetaHi, float maxVoltage, float
    ↪ Vdc) {
        return thetaHi/2+thetaLo/2+(maxVoltage*(cos(thetaHi)-cos(thetaLo)))/
            ↪ Vdc;
        //return output;
}
float addHarmonicCompensation(float thetaK, float thetaLo, float thetaHi,
    ↪ float h, float hc, float Vdc) {
        thetaK += (hc*(cos(h*thetaLo)-cos(h*thetaHi)))/(h*Vdc);
```

```
        return thetaK;
}
void copyAngleSet(switchingTimestamp** fromHead, switchingTimestamp* beginP,
    ↪ switchingTimestamp** toHead, switchingTimestamp** traverser) {
        if(*fromHead == NULL) return;
        switchingTimestamp* current = beginP;
        addswitchingTimestamp(toHead, current->timestamp, current->voltage);
        *traverser = *toHead;

        while(current->prev != NULL) {
                addswitchingTimestamp(toHead, current->timestamp, current->
                    ↪ voltage);
                current = current->prev;
        }

}
void quadrupleAngleSet(switchingTimestamp** head) {
        if(*head == NULL) return;
        switchingTimestamp* current = *head;

        while(current->next != NULL) {
                current = current->next;
                addswitchingTimestamp(head, M_PI - current->timestamp, -
                    ↪ current->voltage);

        }
        current = *head;

        while(current->next != NULL) {
                current = current->next;

                if(current->voltage == current->prev->voltage) {
                        switchingTimestamp* current2 = *head;
                        current2->voltage = -current->prev->voltage;
                }
                addswitchingTimestamp(head, 2*M_PI - current->timestamp,
                    ↪ current->voltage);
                //current = current->next;
        }
}
// Function to calculate pwm sequence for coming period
```

```
int calculateComingPeriod(switchingTimestamp** head, switchingTimestamp**
    ↪ beginP, float maxVoltage, float Vdc, float comingHarmonics[], bool
    ↪ alternateMode, bool FirstMoveState) {
        //FirstMoveState = 1;
        if(!FIRSTMOVE_ALTERING) FirstMoveState = 0;
        if(maxVoltage == 0) return -1;
        if(*head == NULL) {
        for(int i=0; i < (int)swAlphas/8; i++) {

                float thetaLo = i*dTheta;//+(float)sign*M_PI;
                float thetaHi = (i+1)*dTheta;//+(float)sign*M_PI;
                if(FirstMoveState) {
                        //float temp = thetaLo;
                        //thetaLo = thetaHi;
                        //thetaHi = temp;
                        float thetaMid = calculateThetaK(thetaHi, thetaLo,
                            ↪ maxVoltage, Vdc);
                        addswitchingTimestamp(head, thetaLo, Vdc/2);
                        if(i==0) {*beginP = *head;
                        //printf("Traverser tstamp: %f\n", (*traverser)->
                            ↪ timestamp);
                        }
                        addswitchingTimestamp(head, thetaMid, -Vdc/2);
                } else {
                        float thetaMid = calculateThetaK(thetaLo, thetaHi,
                            ↪ maxVoltage, Vdc);
                        addswitchingTimestamp(head, thetaLo, -Vdc/2);
                        if(i==0) {*beginP = *head;
                        //printf("Traverser tstamp: %f\n", (*traverser)->
                            ↪ timestamp);
                        }
                        addswitchingTimestamp(head, thetaMid, Vdc/2);
                }

        }

        if(FirstMoveState) {
                addswitchingTimestamp(head, ((float)0.5)*M_PI, -Vdc/2);
        } else {
                addswitchingTimestamp(head, ((float)0.5)*M_PI, Vdc/2);
        }
```

```
        //swAlphas[SWALPHAS] = (float)2*M_PI;
        } else {

                /*for(int i = 0; i <= SWALPHAS-2; i=i+2) {

                        float thetaLo = *swAlphas[i];
                        float thetaHi = *swAlphas[i+2];

                } */

                switchingTimestamp* current = *head;
                if(current->next == NULL || current->next->next == NULL) {
                        return -1;
                }
                //alternateMode = 0;
                //printf("here ");
                if(alternateMode) current = current->next;
                //printf("here2 ");
                while(1) {
                        if(current->next == NULL || current->next->next == NULL
                            ↪ ) {
                                break;
                        }

                        float thetaHi = current->timestamp;
                        float thetaLo = current->next->next->timestamp;
                        if(FirstMoveState) {
                                float temp = thetaHi;
                                thetaHi = thetaLo;
                                thetaLo = temp;
                        }
                        if(alternateMode) {
                                float temp = thetaHi;
                                thetaHi = thetaLo;
                                thetaLo = temp;
                        }

                        current = current->next;
                        float thetaMid = current->timestamp;

                        int enabledFeatures = 0;
                        //float prevThetaMid = thetaMid;
```

```
if(SHE_FUNDAMENTAL) {
        thetaMid = addHarmonicCompensation(thetaMid,
            ↪ thetaLo, thetaHi, 1, comingHarmonics[
            ↪ enabledFeatures]-maxVoltage, Vdc);
        enabledFeatures++;
}
if(SHE_3H_INJECTION) {
        thetaMid = addHarmonicCompensation(thetaMid,
            ↪ thetaLo, thetaHi, 3, comingHarmonics[
            ↪ enabledFeatures]-maxVoltage/6, Vdc);
        enabledFeatures++;
}

for(int j = enabledFeatures; j < SHE_arrSize; j++) {
        thetaMid = addHarmonicCompensation(thetaMid,
            ↪ thetaLo, thetaHi, HCsSHE[j-
            ↪ enabledFeatures], comingHarmonics[j], Vdc
            ↪ );
}


//float change = 0.05;

/*if(prevThetaMid != 0) {
        if ((signF(thetaMid-prevThetaMid)/prevThetaMid)
            ↪  > change) thetaMid = prevThetaMid+signF(
            ↪ thetaMid-prevThetaMid)*prevThetaMid*
            ↪ change;

}*/
//--------------- Dont overlap switching angles
    ↪ ----------------
if(current->prev != NULL && thetaMid > current->prev->
    ↪ timestamp) {
        thetaMid = current->prev->timestamp;
}
if(current->next != NULL && thetaMid < current->next->
    ↪ timestamp) {
        thetaMid = current->next->timestamp;
}
//--------------- end Comment ----------------
current->timestamp = thetaMid;
```

```
                        current = current->next;
                }
        }

        /*switchingTimestamp* current = *head;
        current = current->next;
        while(current != NULL) {
                addswitchingTimestamp(head, 2*M_PI-current->timestamp, current
                    ↪ ->voltage, NULL);
                current = current->next;
        }*/
        return 1;
}
void emptyList(switchingTimestamp** head_ref)
{

  switchingTimestamp* current = *head_ref;
  switchingTimestamp* next;
  while (current != NULL) {
    next = current->next;
    free(current);
    current = next;
  }
  if(*head_ref != NULL) *head_ref = NULL;
}
```

D.0.2 Start of the code

```
printf("dTheta:␣%f\n", dTheta);

swAlphas = InputSignal(pIn_swAlphas, 0);
dTheta = ((2*M_PI)/((float)swAlphas/2));
motor.polePairs = inPolePairs;
motor.rotorSpeed = inRotorSpeed;
motor.rotorAngle = inRotorAngle;

pwm.F = inFpwm;
pwm.tolerance = PWMTOL;
if(pwm.F > 0) {
 pwm.T = 1/pwm.F;
}
```

```
for(int s = 0; s < N_PHASES; s++) {
        //phase[s] = NULL;
        printf("Init␣Loop␣%d\n", s);
        phase[s].stateReset = true;
        phase[s].stateReset2 = true;
        phase[s].phase = 2*M_PI;

        phase[s].alphaTraverser = NULL;

        for(int i = 0; i < sizeof(HCs)/sizeof(HCs[0]); i++) {
                OutputSignal(pOut_HCsVa+s, i) = 0;
                printf("HC␣resetter:␣%d\n", i);
        }
}
printf("SIZE:␣%d,␣size␣extra:␣%d\n", (int)sizeof(HCs)/(int)sizeof(HCs[0]),
    ↪ SHE_FUNDAMENTAL + SHE_3H_INJECTION);
outTime = 0;

OutputSignal(pOut_SWALPHAS, 0) = swAlphas;
```

D.0.3 Updating part of the code

```
    // Main
Vdc = inVdc;
VdcHalf = Vdc/2;
threshold = inThreshold;
time = (float)inTime;

outTime = time;
motor.rotorSpeed = inRotorSpeed;
motor.rotorAngle = inRotorAngle;

for(int n = 0; n < N_PHASES; n++) {

//------------- phase n calculations ---------------//
phase[n].prevVoltage = phase[n].voltage;
phase[n].voltage = InputSignal(pIn_Vpn, n);

VdGoalLast = VdGoal;
VqGoalLast = VqGoal;
```

```
VdGoal = InputSignal(pIn_VdqGoal, 0);
VqGoal = InputSignal(pIn_VdqGoal, 1);

// float temp = (float)inVaGoal;
/*if(fabs(temp-phase[n].prevGoalVoltage) > 0) {

}*/
phase[n].prevGoalVoltage = phase[n].goalVoltage;
phase[n].goalVoltage = InputSignal(pIn_VnGoal, n);

if (phase[n].voltage >= threshold) {
  phase[n].halfBridgeState = 1;
} else {
  phase[n].halfBridgeState = -1;
}

if(time >= phase[n].nextVSampleTimestamp) {
        sampledPhases++;
        if(sampledPhases >= 3) {
                phase[n].nextVSampleTimestamp = time+constrain(2*M_PI/motor.
                    ↪ rotorSpeed/VF_SAMPLES_PP, 0, VF_MIN_SAMPLE_T);
                sampledPhases = 0;
        }
        phase[n].VSamples[phase[n].samplesIndex] = inRotorSpeed;
        if(phase[n].samplesIndex < VF_SAMPLES_TOT-1) phase[n].samplesIndex++;
        else if(!VF_LIVE) phase[n].samplesIndex = 0;
        else if(VF_LIVE) phase[n].rotorSpeed = averageMotorSpeedsLive(phase[n
            ↪ ].VSamples, phase[n].samplesIndex);
        OutputSignal(pOut_Speed, n) = phase[n].rotorSpeed;
        //printf("Averaged speed: %f, @%fs\n", phase[n].rotorSpeed, time);
}



phase[n].lastPhase = phase[n].phase;

phase[n].phase = InputSignal(pIn_FPIPLL, n);
OutputSignal(pOut_Phase, n) = phase[n].phase;

if (fabs(phase[n].rotorSpeed) > 0 && phase[n].listSize > 0) {

        phase[n].dThetasPassed = (int)(phase[n].phase / dTheta);
```

```
if (abs(phase[n].dThetasPassed - phase[n].dThetasPassedLast) > 0) {
  float thetaLo = (float)dTheta * phase[n].dThetasPassed;
  float thetaHi = (float)dTheta * (phase[n].dThetasPassed + 1);

      if(phase[n].firstMoveState && FIRSTMOVE_ALTERING) {
             float temp = thetaLo;
             thetaLo = thetaHi;
             thetaHi = temp;
      }
  phase[n].thetaMid = calculateThetaK(thetaLo, thetaHi, phase[n].maxVoltage
      ↪ , Vdc);

  phase[n].dThetasPassedLast = phase[n].dThetasPassed;
}
// printf("just before problem");

// printf("ok state: %d", ok);

      //phase[n].alphaTraverser = NULL;

if (SHE) { //if SHE
bool going = true;

      while(going) {
             going = false;
             if (phase[n].alphaTraverser != NULL && phase[n].alphaTraverser
                 ↪ ->prev != NULL) {
             //printf("TRAVERSER HAS VALUES!!\n");
                    if (phase[n].phase >= phase[n].alphaTraverser->prev->
                        ↪ timestamp) {
                    //outModulator1 = signF(phase[n].alphaTraverser->prev->
                        ↪ voltage);
                    phase[n].alphaTraverser = phase[n].alphaTraverser->prev
                        ↪ ;
             going = true;
             }

             }
             if (phase[n].alphaTraverser != NULL && phase[n].alphaTraverser
                 ↪ ->next != NULL) {
             //printf("TRAVERSER HAS VALUES!!\n");
```

```
                        if (phase[n].phase < phase[n].alphaTraverser->timestamp
                            ↪ ) {
                        //OutputSignal(pOut_Mod, n) = signF(phase[n].
                            ↪ alphaTraverser->next->voltage);
                        phase[n].alphaTraverser = phase[n].alphaTraverser->next
                            ↪ ;
                going = true;
                }

                }
        }
        if(phase[n].alphaTraverser != NULL) {
                OutputSignal(pOut_Mod, n) = signF(phase[n].alphaTraverser->
                    ↪ voltage);
   }
  } else {
    if (phase[n].phase >= phase[n].thetaMid) {
      OutputSignal(pOut_Mod, n) = 1;
    } else {
      OutputSignal(pOut_Mod, n) = -1;
    }
    if(phase[n].firstMoveState && FIRSTMOVE_ALTERING) OutputSignal(pOut_Mod,
        ↪ n) = -1*OutputSignal(pOut_Mod, n);
  }
  // printf("after");
  //}
  //*/
} else {
        OutputSignal(pOut_Phase, n) = 0;
  //outPhaseVa = 0;
  OutputSignal(pOut_Mod, n) = 0;
}

if ((phase[n].phase > (float)0.5 * M_PI && phase[n].stateReset)) {
  phase[n].state = signF(phase[n].goalVoltage);
  phase[n].stateReset = false;

  //printf("phase state: %d @%fs, phase: %f\n", phase[n].state, time, phase[
      ↪ n].phase);
}
if (phase[n].phase > (float)1.5 * M_PI && phase[n].stateReset2) {
  phase[n].state = signF(phase[n].goalVoltage);
```

```
  phase[n].stateReset2 = false;
  //printf("phase state: %d @%fs, phase: %f\n", phase[n].state, time, phase[
      ↪ n].phase);
}


//*/
/*if(time-maxVoltageTimer > 1/1000) {

}*/
// update VPN's on every rising/falling edge of the square wave

if ((phase[n].voltage) > threshold && (phase[n].prevVoltage) <= threshold &&
    ↪  (time - phase[n].lastRiseTimestamp) > pwm.T * pwm.tolerance && phase[
    ↪ n].modState == false) {
  // printf("rising edge\n");
  phase[n].modState = true;
  phase[n].lastRiseTimestamp = time;
  addswitchingTimestamp(&switchingTimestampsVPN[n], time, Vdc / 2);
  phase[n].listSize++;
  switchingTimestamp* current = switchingTimestampsVPN[n];
  if (current == NULL) {
    goto skipAvg;
  }
  float timestamp = current->timestamp;
  current = current->next;
  if (current == NULL) {
    goto skipAvg;
  }
  float earlierTimestamp = current->timestamp;
  double voltage = current->voltage;
  current = current->next;
  if (current == NULL) {
    goto skipAvg;
  }
  float earliererTimestamp = current->timestamp;
  double voltage2 = current->voltage;
  phase[n].lastPeriodAverage = phase[n].periodAverage;
  phase[n].periodAverage = (earlierTimestamp - earliererTimestamp) / (
      ↪ timestamp - earliererTimestamp) * voltage2 + (timestamp -
      ↪ earlierTimestamp) / (timestamp - earliererTimestamp) * voltage;
```

```
  // printf("time: %f, voltage averaged: %f, last one: %f, v1: %f, v2: %f,
     ↪ t1: %f, t2: %f, t3: %f\n", time, phase[n].periodAverage, phase[n].
     ↪ lastPeriodAverage, voltage, voltage2, earliererTimestamp,
     ↪ earlierTimestamp, timestamp);
  goto doneAvg;

skipAvg:

  phase[n].periodAverage = 0;
  phase[n].lastPeriodAverage = 0;
  //printf("empty list!\n");

  // outPeriodUpdate = 0;

  // riseFallDetect = 1;
} else if ((phase[n].voltage) < threshold && (phase[n].prevVoltage) >=
    ↪ threshold && (time - phase[n].lastFallTimestamp) > pwm.T * pwm.
    ↪ tolerance && phase[n].modState == true) {
  // printf("falling edge\n");
  phase[n].modState = false;
  phase[n].lastFallTimestamp = time;
  addswitchingTimestamp(&switchingTimestampsVPN[n], time, -Vdc / 2);
  phase[n].listSize++;

  // outPeriodUpdate = 0;
}

doneAvg :


if(time-sheUpdateTime > (float)1/10000) {
        sheUpdatedPhases++;
        if(sheUpdatedPhases >= 3) {
                sheUpdateTime = CurrentTime;
                sheUpdatedPhases = 0;
        }
        phase[n].maxVoltage = sqrt(pow(VdGoal, 2) + pow(VqGoal, 2));
        //printf("SHE maxV: %f\n", phase[n].maxVoltage);

        for (int i = 0; (i < SHE_ITERATIONS && SHE && !completedSHE(phase[n].
            ↪ comingHarmonics, phase[n].maxVoltage)); i++) {
```

```
    calculateComingPeriod(&comingStampsQuartV[n], &comingStampsQuartBeginV[n
        ↪ ], phase[n].maxVoltage, Vdc, phase[n].comingHarmonics, 0, phase[n
        ↪ ].firstMoveState);
    calculateComingHarmonics(comingStampsQuartV[n], phase[n].comingHarmonics)
        ↪ ;

        /*if(n==1) {
    printf("iterative SHE (phase %d, i: %d): ", n, i);
    for (int j = 0; j < SHE_arrSize; j++) {
      printf("h_%d=%.2f ", j + 1, phase[n].comingHarmonics[j]);
    }
    printf("\n");
    }*/
/*}
    switchingTimestamp* current = comingSwitchingTimestampsV[n];
    printf("switching alphas:\n");
    while (current != NULL) {
      printf("%f, ", current->timestamp);
      current = current->next;
    }
                printf("\n");*/

        if(THETAS_SWITCHING_WITH_SIGMAS) {
    calculateComingPeriod(&comingStampsQuartV[n], &comingStampsQuartBeginV[n
        ↪ ], phase[n].maxVoltage, Vdc, phase[n].comingHarmonics, 1, phase[n
        ↪ ].firstMoveState);
    // printf("here");
    calculateComingHarmonics(comingStampsQuartV[n], phase[n].comingHarmonics)
        ↪ ;
  }

        /*current = comingSwitchingTimestampsV[n];
    printf("switching alphas:\n");
    while (current != NULL) {
      printf("%f, ", current->timestamp);
      current = current->next;
    }
    printf("\n");*/

    if(n==1) {
    printf("iterative␣SHE␣(phase␣%d,␣i:␣%d):␣", n, i);
    for (int j = 0; j < SHE_arrSize; j++) {
```

```
      printf("h_%d=%.2f␣", j + 1, phase[n].comingHarmonics[j]);
    }
    printf("\n");
    }

  }
  if(SHE) {
  emptyList(&comingStampsFullV[n]);
  copyAngleSet(&comingStampsQuartV[n], comingStampsQuartBeginV[n], &
      ↪ comingStampsFullV[n], &phase[n].alphaTraverser);
  quadrupleAngleSet(&comingStampsFullV[n]);

  /*
  if(n==0) {
  switchingTimestamp* test = phase[n].alphaTraverser;
  while(test != NULL) {
        printf("TRAVERSER phase %d tstamp 'returned': %f, voltage: %f, maxV:
            ↪ %f\n", n, test->timestamp, test->voltage, phase[n].maxVoltage);
        test = test->prev;
  }
  printf("-----------------------------------------------------\n");
  }*/
  }
}


if (phase[n].listSize >= 4 && phase[n].state > 0 && phase[n].phase > 1.5 *
    ↪ M_PI && phase[n].stateReset == false && phase[n].stateReset2 == false)
    ↪  {
  phase[n].state = signF(InputSignal(pIn_VnGoal, n));

  // printf("state resetter tripped!");
}

//----------- print variables at slow pace
if(time-slowPrinter > (float)1/5000) {
      //printf("error ed: %f, eq: %f\n", inEd, inEq);
       //printf("var1: %.8f, var2: %d, var3: %.20f, var4: %.20f, var5: %d\n
           ↪ ", time, phase[n].state, phase[n].goalVoltage, phase[n].
           ↪ prevGoalVoltage, phase[n].listSize);
       /*if(phase[n].alphaTraverser != NULL && phase[n].alphaTraverser->prev
           ↪  != NULL && phase[n].alphaTraverser->prev->prev != NULL) {
```

```
        slowPrinter = time;
        printf("SHE DEBUG: phase: %f, t1: %f, t2: %f, t3: %f\n", phase[n].
            ↪ phase, phase[n].alphaTraverser->timestamp, phase[n].
            ↪ alphaTraverser->prev->timestamp, phase[n].alphaTraverser->prev
            ↪ ->prev->timestamp);
                }//*/


                //if(phase[n].alphaTraverser!=NULL) printf("Debug phase %d;
                    ↪ NULL??: %d\n", n, phase[n].alphaTraverser==NULL);
                //if(phase[n].alphaTraverser != NULL) printf(" next element: %
                    ↪ d", phase[n].alphaTraverser->next==NULL);
                //printf("\n");
                //phase[n].alphaTraverser = NULL;
                slowPrinter = time;
                //printf("Debug phase %d; NULL??: %d\n", n, phase[n].
                    ↪ alphaTraverser==NULL);
                //printf("debug phase %d: G: %f, pG: %f, lsize: %d, %f, %d,
                    ↪ cMode: %d\n", n, phase[n].goalVoltage, phase[n].
                    ↪ prevGoalVoltage, phase[n].listSize, motor.rotorSpeed,
                    ↪ phase[n].stateReset, calcMode);
                //printf("debug: last rise stamp %f, %f, %d, %f, %d\n", signF(
                    ↪ phase[n].goalVoltage), signF(phase[n].prevGoalVoltage),
                    ↪  phase[n].listSize, motor.rotorSpeed, phase[n].
                    ↪ stateReset);
}

//if ( ((signF(phase[n].goalVoltage) != signF(phase[n].prevGoalVoltage) &&
    ↪ calcMode == 2) || (signF(phase[n].goalVoltage) == 1 && signF(phase[n].
    ↪ prevGoalVoltage) == -1 && calcMode == 1)) &&
//phase[n].listSize >= 4 && motor.rotorSpeed > 0 && !phase[n].stateReset) {
if(phase[n].phase < 0.25*M_PI && phase[n].lastPhase > 1.75*M_PI && !phase[n
    ↪ ].stateReset && phase[n].listSize >= 4) {

  addswitchingTimestamp(&switchingTimestampsVPN[n], time, 0);
  float harmonics[sizeof(HCs)/sizeof(HCs[0])];
  phase[n].deltaT = calculateHarmonics(Vdc, switchingTimestampsVPN[n],
      ↪ harmonics, phase[n].beginTimePeriod, calcMode, !phase[n].
      ↪ comingNegativePeriod);
  // Print or use the calculated harmonics for VPN1
  /*printf("Vpn%d harmonics: ", n);
  for (int j = 0; j < sizeof(HCs)/sizeof(HCs[0]); j++) {
```

```
    printf("h_%d=%.2f ", HCs[j], harmonics[j]);
  }
  printf("\n");
  */

for(int i = 0; i < sizeof(HCs)/sizeof(HCs[0]); i++) {
       OutputSignal((int)pOut_HCsVa+n, i) = harmonics[i];
}


  emptyList(&switchingTimestampsVPN[n]);
  phase[n].listSize = 1;
  addswitchingTimestamp(&switchingTimestampsVPN[n], time, -Vdc / 2);

       phase[n].firstMoveState = randnum(0,1);
       //phase[n].firstMoveState = 1;
  phase[n].maxVoltage = sqrt(pow(InputSignal(pIn_VdqGoal, 0), 2) + pow(
     ↪ InputSignal(pIn_VdqGoal, 1), 2));


  if(phase[n].state == -1) {
       phase[n].comingNegativePeriod = 0;
  } else {
       phase[n].comingNegativePeriod = 1;
  }

  //printf("max3phV: %f, Va*: %f, Va*prev: %f, First Move State: %d\n",
     ↪ phase[n].maxVoltage, phase[n].goalVoltage, phase[n].prevGoalVoltage,
     ↪  phase[n].firstMoveState);
  //int arrayLength = sizeof(phase[n].comingHarmonics) / sizeof(phase[n].
     ↪ comingHarmonics[0]);
  /*for (int i = 0; i < arrayLength; i++) {
    phase[n].comingHarmonics[i] = 0;
  }*/

  phase[n].beginTimePeriod = time;

  if(!VF_LIVE) phase[n].rotorSpeed = averageMotorSpeeds(phase[n].VSamples, &
     ↪ phase[n].samplesIndex, motor.rotorSpeed, &phase[n].
     ↪ nextVSampleTimestamp);
```

```
  phase[n].phase = 0;
  phase[n].stateReset = true;
  phase[n].stateReset2 = true;
  //phase[n].state = 1;

  OutputSignal(pOut_PeriodUpdate, n) = 1;
  phase[n].lastPeriodTimestamp = time;
  phase[n].lastHalfPeriodTime = time - phase[n].lastHalfPeriodTimestamp;
  phase[n].lastHalfPeriodTimestamp = time;


} else if (time - phase[n].lastPeriodTimestamp >= (float)1 / 100000) {
  OutputSignal(pOut_PeriodUpdate, n) = 0;
}

}

return;
```

Appendix E

Code of the MPC algorithm

E.0.1 Definitions and Functions of the code

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#define SAMPLETIME (float)1/100000
float Ts = SAMPLETIME;

#define pIn_Mconsts 0
#define pIn_iGoal  1
#define pIn_i   2
#define pIn_Uemf  3
#define pIn_mSpeedAngle 4
#define pIn_idqGoal 5
#define pIn_Vdc  6

#define pOut_cost  0
#define pOut_Uab  1
#define pOut_Udq  2
#define pOut_Uemf  3

#define COST_OPTIMIZED_THRESHOLD 0.005
#define GD_ITERATIONS 125
#define PREDICTIONS 11
#define vLength  2
#define LRATE   10
#define PREV_I_SAMPLES 2
#define PHASES   3
```

```
#define POSSIBLE_VECTORS 8

typedef struct predictionStruct {
	float ip[vLength][PREDICTIONS], i[vLength][PREDICTIONS], iPrev[
	    ↪ vLength][PREV_I_SAMPLES], iGoalFurthestHorizon[vLength], idGoal
	    ↪ , iqGoal,
	U[vLength][PREDICTIONS], Vd, Vq, VdTemp, VqTemp, Uemf[vLength][
	    ↪ PREDICTIONS], totCost, prevCost[vLength], leastCost[PREDICTIONS
	    ↪ ];
	int leastCostIndex[PREDICTIONS];
} predictionStruct;
predictionStruct p = { 0 };

typedef struct motorStruct {
	float L[vLength], R, angle, speed;
} motorStruct;
motorStruct M = { 0 };

float Vdc = 0;
float Valbe[vLength][POSSIBLE_VECTORS] = { 0 };
float VhBridge[2];
//float Vbe[POSSIBLE_VECTORS] = { 0 };

float invParkAlpha(float d, float q, float a) {

	return d*cos(a)-q*sin(a);
}

float invParkBeta(float d, float q, float a) {

	return d*sin(a)+q*cos(a);
}

float ClarkeAlpha(float Vabc[]) {

	return (float)2/(float)3*(Vabc[0]-0.5*Vabc[1]-0.5*Vabc[2]);
}
float ClarkeBeta(float Vabc[]) {
	//printf("sqrt(3): %f", Vabc[0]);
	return (float)2/(float)3*(sqrtf(3)/(float)2*Vabc[1]-sqrtf(3)/(float)
	    ↪ 2*Vabc[2]);
}
```

```
float signF(float num) {
       if(num < 0) {
              return -1;
       } else {
              return 1;
       }
}
/**
 * Find maximum between two numbers.
 */
float max(float num1, float num2)
{
    return (num1 > num2 ) ? num1 : num2;
}

/**
 * Find minimum between two numbers.
 */
float min(float num1, float num2)
{
    return (num1 > num2 ) ? num2 : num1;
}

float constrain(float num, float lo, float hi) {
       return max(min(num, hi), lo);
}

float avg(float num1, float num2) {

       return (num1+num2)/2;

}
```

E.0.2 Start of the code

```
M.L[0] = InputSignal(pIn_Mconsts, 0);
M.L[1] = InputSignal(pIn_Mconsts, 1);
M.R = InputSignal(pIn_Mconsts, 2);
Vdc = InputSignal(pIn_Vdc, 0);
Vdc = 400;
```

```
VhBridge[0] = -Vdc/2;
VhBridge[1] = Vdc/2;

int hBridgeStates = 2;
int a = 0;
for(int i = 0; i < hBridgeStates; i++) {
       for(int j = 0; j < hBridgeStates; j++) {
              for(int k = 0; k < hBridgeStates; k++) {
                     float Vabc[PHASES];
                     Vabc[0] = VhBridge[i];
                     Vabc[1] = VhBridge[j];
                     Vabc[2] = VhBridge[k];

                     Valbe[0][a] = ClarkeAlpha(Vabc);
                     Valbe[1][a] = ClarkeBeta(Vabc);

                     printf("i:␣%d,␣j:␣%d,␣k:␣%d,␣alpha:␣%f,␣beta:␣%f\n", i,
                         ↪  j, k, Valbe[0][a], Valbe[1][a]);
                     //printf("Vabc: %f, %f, %f, Vdc: %f\n", Vabc[0], Vabc
                         ↪ [1], Vabc[2], Vdc);
                     a++;
              }
       }
}
```

E.0.3 Updating part of the code

```
M.speed = InputSignal(pIn_mSpeedAngle, 0);
M.angle = InputSignal(pIn_mSpeedAngle, 1);

Vdc = InputSignal(pIn_Vdc, 0);

p.idGoal = InputSignal(pIn_idqGoal, 0);
p.iqGoal = InputSignal(pIn_idqGoal, 1);


//for(int a = 0; a < POSSIBLE_VECTORS; a++) {


/*
```

```
for(int j = 0; j < PREDICTIONS; j++) {
p.leastCost[j] = INFINITY;
for(int a = 0; a < POSSIBLE_VECTORS; a++) {

p.totCost = 0;

for(int i = 0; i < vLength; i++) {
        if(j > 0) {
                p.i[i][j] = p.ip[i][j-1];
                //p.i[i][j] = 3*p.ip[i][j-1]-3*p.iPrev[i][j-1]+p.iPrev[
                    ↪ i][j];
        } else {
                p.i[i][j] = InputSignal(pIn_i, i);
        }
        if(i==0) {
                p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*-sin(M.angle+j*
                    ↪ M.speed*Ts);
                //p.U[i][j] = invParkAlpha(p.Vd, p.Vq, M.angle+j*M.
                    ↪ speed*Ts);
                p.U[i][j] = Valbe[i][a];
                p.iGoalFurthestHorizon[i] = invParkAlpha(p.idGoal, p.
                    ↪ iqGoal, M.angle+PREDICTIONS*M.speed*Ts);

                //printf("Prediction: %d, wIters: %d, iGoal: %f, U: %f,
                    ↪  Uemf: %f\n", j, wIters, p.iGoal[i][j], p.U[i][j
                    ↪ ], p.Uemf[i][j]);
        } else {
                p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*cos(M.angle+j*M
                    ↪ .speed*Ts);
                //p.U[i][j] = invParkBeta(p.Vd, p.Vq, M.angle+j*M.speed
                    ↪ *Ts);
                p.U[i][j] = Valbe[i][a];
                p.iGoalFurthestHorizon[i] = invParkBeta(p.idGoal, p.
                    ↪ iqGoal, M.angle+PREDICTIONS*M.speed*Ts);
        }

        p.ip[i][j] = (1-M.R*Ts/M.L[i])*p.i[i][j] + Ts/M.L[i]*(p.U[i][j
            ↪ ] - p.Uemf[i][j]);

        p.totCost += powf(p.iGoalFurthestHorizon[i]-p.ip[i][
            ↪ PREDICTIONS-1], 2);
}
```

```
        printf("p.totCost: %f\n", p.totCost);
                if(p.totCost < p.leastCost[j]) {

                        p.leastCost[j] = p.totCost;
                        p.leastCostIndex[j] = a;

                        printf("horizon: %d, leastCost: %f, vectorIndex: %d\n",
                            ↪  j, p.leastCost[j], p.leastCostIndex[j]);

                }

        }

}

        */
//}

int wIters = 0;
while((p.totCost > COST_OPTIMIZED_THRESHOLD && wIters < GD_ITERATIONS) ||
    ↪ wIters == 0) {
        p.totCost = 0;

        for(int i = 0; i < vLength; i++) {
        for(int j = 0; j < PREDICTIONS; j++) {


                if(j > 0) {
                        p.i[i][j] = p.ip[i][j-1];
                        //p.i[i][j] = 3*p.ip[i][j-1]-3*p.iPrev[i][j-1]+p.iPrev[
                            ↪ i][j];
                } else {
                        p.i[i][j] = InputSignal(pIn_i, i);
                }
                if(i==0) {
                        p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*-sin(M.angle+j*
                            ↪ M.speed*Ts);
                        p.U[i][j] = invParkAlpha(p.Vd, p.Vq, M.angle+j*M.speed*
                            ↪ Ts);
                        p.iGoalFurthestHorizon[i] = invParkAlpha(p.idGoal, p.
                            ↪ iqGoal, M.angle+PREDICTIONS*M.speed*Ts);
```

```
                //printf("Prediction: %d, wIters: %d, iGoal: %f, U: %f,
                    ↪  Uemf: %f\n", j, wIters, p.iGoal[i][j], p.U[i][j
                    ↪ ], p.Uemf[i][j]);
            } else {
                p.Uemf[i][j] = InputSignal(pIn_Uemf, 0)*cos(M.angle+j*M
                    ↪ .speed*Ts);
                p.U[i][j] = invParkBeta(p.Vd, p.Vq, M.angle+j*M.speed*
                    ↪ Ts);
                p.iGoalFurthestHorizon[i] = invParkBeta(p.idGoal, p.
                    ↪ iqGoal, M.angle+PREDICTIONS*M.speed*Ts);
            }

            p.ip[i][j] = (1-M.R*Ts/M.L[i])*p.i[i][j] + Ts/M.L[i]*(p.U[i][j
                ↪ ] - p.Uemf[i][j]);

            //if(p.totCost
            //printf("wIter: %d, i: %d, j: %d, iGoal: %f, ip: %f, Cost:
                ↪ %.3f, Ufound: %f, emf: %f\n", wIters, i, j, p.
                ↪ iGoalFurthestHorizon[i], p.ip[i][j], p.totCost, p.U[i][
                ↪ i], p.Uemf[i][i]);
            //p.AuTemp = p.AuTemp - 2*(InputSignal(pIn_iGoal, i)-p.ip[i][
                ↪ PREDICTIONS-1])*
        //Ts/M.L[i]*(-sin(M.angle+(PREDICTIONS-1)*M.speed*Ts)*(float)i==0+cos
            ↪ (M.angle+(PREDICTIONS-1)*M.speed*Ts)*(float)i==1);
        }

            p.totCost += powf(p.iGoalFurthestHorizon[i]-p.ip[i][
                ↪ PREDICTIONS-1], 2);
            if(i==0) {
                p.VdTemp = p.VdTemp + LRATE*2*(p.iGoalFurthestHorizon[i
                    ↪ ]-p.ip[i][PREDICTIONS-1])*Ts/M.L[i]*cos(M.angle
                    ↪ +(PREDICTIONS-1)*M.speed*Ts);
                p.VqTemp = p.VqTemp + LRATE*2*(p.iGoalFurthestHorizon[i
                    ↪ ]-p.ip[i][PREDICTIONS-1])*Ts/M.L[i]*-sin(M.angle
                    ↪ +(PREDICTIONS-1)*M.speed*Ts);
            } else {
                p.VdTemp = p.VdTemp + LRATE*2*(p.iGoalFurthestHorizon[i
                    ↪ ]-p.ip[i][PREDICTIONS-1])*Ts/M.L[i]*sin(M.angle
                    ↪ +(PREDICTIONS-1)*M.speed*Ts);
                p.VqTemp = p.VqTemp + LRATE*2*(p.iGoalFurthestHorizon[i
                    ↪ ]-p.ip[i][PREDICTIONS-1])*Ts/M.L[i]*cos(M.angle
                    ↪ +(PREDICTIONS-1)*M.speed*Ts);
```

```
                }

        }

        p.Vd = p.VdTemp;
        p.Vq = p.VqTemp;
        /*if(i==0) {

                p.V

        } else {
                p.cost[i] = powf(InputSignal(pIn_iGoal, i)-p.ip[i][PREDICTIONS
                    ↪ -1], 2);
                p.U[i][0] = p.U[i][0] - 2*(InputSignal(pIn_iGoal, i)-p.ip[i][
                    ↪ PREDICTIONS-1])*Ts/M.L[i];
        }*/
        wIters++;
}

printf("D; wIters: %d, iGoal: %f, ip: %f, i: %f, Cost: %.3f, Ufound: %f, emf
    ↪ : %f\n", wIters, p.iGoalFurthestHorizon[0], p.ip[0][PREDICTIONS-1], p.
    ↪ i[0][PREDICTIONS-1], p.totCost, p.U[0][PREDICTIONS-1], p.Uemf[0][
    ↪ PREDICTIONS-1]);
printf("Q; wIters: %d, iGoal: %f, ip: %f, i: %f, Cost: %.3f, Ufound: %f, emf
    ↪ : %f\n", wIters, p.iGoalFurthestHorizon[1], p.ip[1][PREDICTIONS-1], p.
    ↪ i[1][PREDICTIONS-1], p.totCost, p.U[1][PREDICTIONS-1], p.Uemf[1][
    ↪ PREDICTIONS-1]);

float out1 = p.U[0][PREDICTIONS-1], out2 = p.U[1][PREDICTIONS-1];

float magnitude = sqrt(powf(p.Vd, 2)+powf(p.Vq, 2));
if(magnitude > Vdc/2) {
        out1 *= Vdc/2/magnitude;
        out2 *= Vdc/2/magnitude;
}

for(int i = 0; i < vLength; i++) {
for(int j = PREV_I_SAMPLES-1; j > 0; j--) {
        p.iPrev[i][j] = p.iPrev[i][j-1];
}
p.iPrev[i][0] = InputSignal(pIn_i, i);
}
```

```
//printf("Magnitude: %f\n", magnitude);
OutputSignal(pOut_cost, 0) = p.totCost;
OutputSignal(pOut_Uab, 0) = out1;
OutputSignal(pOut_Uab, 1) = out2;
OutputSignal(pOut_Udq, 0) = p.Vd;
OutputSignal(pOut_Udq, 1) = p.Vq;
OutputSignal(pOut_Uemf, 0) = p.Uemf[0][0];
OutputSignal(pOut_Uemf, 1) = p.Uemf[1][0];

//printf("wIters: %d, Cost: %.3f, Ufound: %f, a emf: %f, b emf: %f\n",
    ↪ wIters, p.totCost, p.U[0][0], p.Uemf[0][0], p.Uemf[0][1]);
```

Appendix F

Code of the LMS-based SHE

F.0.1 Definitions and Functions of the code

```
#define pIn_time   0
#define pIn_PLL   1
#define pIn_I    2
#define pIn_SWALPHAS  3
#define pIn_MPP   4

#define pOut_fundLMS  0
#define pOut_funde  1
#define pOut_LMS   2
#define pOut_Uashes  3
#define pOut_HCmag  4

#define N_PHASES    3
#define SAMPLE_FREQ    500000

#define MU     0.0006
//int HCsSHE[] = {1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43,
    ↪ 47, 49, 53, 55, 59, 61, 65, 67, 71, 73, 77, 79, 83, 85, 89, 91};
//int HCsSHE[] = {1, 5, 7, 11, 13, 17, 19, 23, 25, 29};
int HCsSHE[] = {1,5, 7, 11, 13};
int sheSize = sizeof(HCsSHE)/sizeof(HCsSHE[0]);

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

float time = 0, mu = MU;
int swAlphas, MPP;
```

```
//double filtCutFreq = 0, decades = 0;

typedef struct lmsStruct {
	float ek[sizeof(HCsSHE)/sizeof(HCsSHE[0])], Wc[sizeof(HCsSHE)/sizeof(
	    ↪ HCsSHE[0])], Ws[sizeof(HCsSHE)/sizeof(HCsSHE[0])], mag[sizeof(
	    ↪ HCsSHE)/sizeof(HCsSHE[0])];
} lmsStruct;
lmsStruct lms[N_PHASES] = {0};

typedef struct asheStruct {
	float phase, Xkc[sizeof(HCsSHE)/sizeof(HCsSHE[0])], Xks[sizeof(HCsSHE
	    ↪ )/sizeof(HCsSHE[0])], out[sizeof(HCsSHE)/sizeof(HCsSHE[0])],
	    ↪ outIP[sizeof(HCsSHE)/sizeof(HCsSHE[0])],
	outIPSummed;
} asheStruct;
asheStruct ashe[N_PHASES];


double signD(double num) {
	if(num < 0) {
		return -1;
	} else {
		return 1;
	}
}

int signI(int num) {
	if(num < 0) {
		return -1;
	} else {
		return 1;
	}
}

float signF(float num) {
	if(num < 0) {
		return -1;
	} else {
		return 1;
	}
}
/**
```

```
 * Find maximum between two numbers.
 */
float max(float num1, float num2)
{
    return (num1 > num2 ) ? num1 : num2;
}

/**
 * Find minimum between two numbers.
 */
float min(float num1, float num2)
{
    return (num1 > num2 ) ? num2 : num1;
}

float constrain(float num, float lo, float hi) {
       return max(min(num, hi), lo);
}

float avg(float num1, float num2) {

       return (num1+num2)/2;

}
```

F.0.2 Start of the code

```
MPP = InputSignal(pIn_MPP, 0);
```

F.0.3 Updating part of the code

```
time = InputSignal(pIn_time, 0);

swAlphas = InputSignal(pIn_SWALPHAS, 0);

for(int i = 0; i < N_PHASES; i++) {
       ashe[i].phase = InputSignal(pIn_PLL,i);

       /*ashe[i].Xkc[0] = cos(ashe[i].phase);
       ashe[i].Xks[0] = sin(ashe[i].phase);
```

```
ashe[i].out[0] = lms[i].Wc[0]*ashe[i].Xkc[0]+lms[i].Ws[0]*ashe[i].Xks
    ↪ [0];
lms[i].ek[0] = InputSignal(pIn_I, i)-ashe[i].out[0];

float temp = atan2f(lms[i].Ws[0], lms[i].Wc[0]);
lms[i].mag[0] = lms[i].Wc[0]*cos(temp)+lms[i].Ws[0]*sin(temp);

lms[i].Wc[0] = lms[i].Wc[0]+2*mu*lms[i].ek[0]*ashe[i].Xkc[0];
lms[i].Ws[0] = lms[i].Ws[0]+2*mu*lms[i].ek[0]*ashe[i].Xks[0];*/
ashe[i].outIPSummed = 0;

for(int j = 0; j < sheSize; j++) {
        ashe[i].Xkc[j] = cos(HCsSHE[j]*ashe[i].phase);
        ashe[i].Xks[j] = sin(HCsSHE[j]*ashe[i].phase);

        ashe[i].out[j] = lms[i].Wc[j]*ashe[i].Xkc[j]+lms[i].Ws[j]*ashe
            ↪ [i].Xks[j];
        ashe[i].outIP[j] = lms[i].Ws[j]*ashe[i].Xkc[j]-lms[i].Wc[j]*
            ↪ ashe[i].Xks[j];
        if(j==0) {
                lms[i].ek[j] = InputSignal(pIn_I, i)-ashe[i].out[j];
        } else {
                lms[i].ek[j] = ashe[i].out[0];
        }

        float temp = atan2f(lms[i].Ws[j], lms[i].Wc[j]);
        lms[i].mag[j] = lms[i].Wc[j]*cos(temp)+lms[i].Ws[j]*sin(temp);

        //if(j==0) {
                lms[i].Wc[j] = lms[i].Wc[j]+2*mu*lms[i].ek[j]*ashe[i].
                    ↪ Xkc[j];
                lms[i].Ws[j] = lms[i].Ws[j]+2*mu*lms[i].ek[j]*ashe[i].
                    ↪ Xks[j];
        //} else {
        // lms[i].Wc[j] = lms[i].Wc[j]+2*mu/100*lms[i].ek[j]*ashe[i].
            ↪ Xkc[j];
        // lms[i].Ws[j] = lms[i].Ws[j]+2*mu/100*lms[i].ek[j]*ashe[i].
            ↪ Xks[j];
        //}

        if(i == 0) {
```

```
                OutputSignal(pOut_HCmag, j) = lms[0].mag[j];
                OutputSignal(pOut_LMS, j) = ashe[0].out[j];

                }
                if(j!=0) {
                        ashe[i].outIPSummed += ashe[i].outIP[j];
                }

        }

        OutputSignal(pOut_Uashes, i) = ashe[i].outIPSummed;

        OutputSignal(pOut_fundLMS, i) = ashe[i].out[0];
        OutputSignal(pOut_funde, i) = lms[i].ek[0];
}

//printf("@: %f, a: %f, cutFreq: %f, Va phase unfiltered: %f, C.cutFreq: %f,
    ↪  C.e: %f, C.P: %f, C.I: %f, C.out: %f, P.out: %f\n", time, filt.a,
    ↪ filt.cutFreq, phase[0], C.cutFreq, C.e[0], C.P[0], C.I[0], C.out[0], P
    ↪ .out[0]);
```

Appendix G

End of life code

This is a previous version of the SHE code, however the accompanying text is still usefull as it can provide insight into how the code works, since parts of the final code match with this code.

G.0.1 Definitions and Functions of the code

Below is the detailed description of the provided code, including the definitions and functions used, explained in a manner aimed at fellow programmers. The code itself is presented in code blocks for clarity.

Header Files The program starts with including standard libraries necessary for mathematical calculations, input/output operations, memory allocation, and boolean operations:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

Macro Definitions Next, several macros are defined to set various parameters and to simplify accessing inputs and outputs:

```
//#define M_PI acos(-1.0)
#define PWMTOL 0.75
#define SWALPHAS 16 //NEEDS to be multiple of 4
#define NUM_HARMONICS 13
#define SHE 1 //set SHE on or off
#define SHE_ITERATIONS 150
```

Input and Output Signal Macros These macros map input and output signals to specific channels for easier reference within the code:

```
#define inVPN1 InputSignal(0,0)
#define inVPN2 InputSignal(0,1)
#define inVPN3 InputSignal(0,2)
```

```
#define inVdc InputSignal(1,0)
#define inThreshold InputSignal(2,0)
#define inTime InputSignal(3,0)
#define inRotorSpeed InputSignal(4,0)
#define inRotorAngle InputSignal(4,1)
#define inPolePairs InputSignal(5,0)
#define inFpwm InputSignal(6,0)
#define inVdGoal InputSignal(7,0)
#define inVqGoal InputSignal(7,1)
#define inVaGoal InputSignal(8,0)
#define inVbGoal InputSignal(8,1)
#define inVcGoal InputSignal(8,2)
#define inPLLPhase InputSignal(9,0)

#define outV1HC1 OutputSignal(0,0)
#define outV1HC2 OutputSignal(0,1)
#define outV1HC3 OutputSignal(0,2)
#define outV1HC4 OutputSignal(0,3)
#define outV1HC5 OutputSignal(0,4)
#define outV1HC6 OutputSignal(0,5)
#define outV1HC7 OutputSignal(0,6)
#define outV1HC8 OutputSignal(0,7)
#define outV1HC9 OutputSignal(0,8)
#define outV1HC10 OutputSignal(0,9)
#define outV1HC11 OutputSignal(0,10)
#define outV1HC12 OutputSignal(0,11)
#define outV1HC13 OutputSignal(0,12)

#define outV2HC1 OutputSignal(1,0)
#define outV2HC2 OutputSignal(1,1)
#define outV2HC3 OutputSignal(1,2)
#define outV2HC4 OutputSignal(1,3)
#define outV2HC5 OutputSignal(1,4)
#define outV2HC6 OutputSignal(1,5)
#define outV2HC7 OutputSignal(1,6)
#define outV2HC8 OutputSignal(1,7)
#define outV2HC9 OutputSignal(1,8)
#define outV2HC10 OutputSignal(1,9)
#define outV2HC11 OutputSignal(1,10)
#define outV2HC12 OutputSignal(1,11)
#define outV2HC13 OutputSignal(1,12)
```

```
#define outV3HC1 OutputSignal(2,0)
#define outV3HC2 OutputSignal(2,1)
#define outV3HC3 OutputSignal(2,2)
#define outV3HC4 OutputSignal(2,3)
#define outV3HC5 OutputSignal(2,4)
#define outV3HC6 OutputSignal(2,5)
#define outV3HC7 OutputSignal(2,6)
#define outV3HC8 OutputSignal(2,7)
#define outV3HC9 OutputSignal(2,8)
#define outV3HC10 OutputSignal(2,9)
#define outV3HC11 OutputSignal(2,10)
#define outV3HC12 OutputSignal(2,11)
#define outV3HC13 OutputSignal(2,12)

#define outModulator1 OutputSignal(3,0)
#define outModulator2 OutputSignal(3,1)
#define outModulator3 OutputSignal(3,2)

#define outPeriodUpdate1 OutputSignal(4,0)
#define outPeriodUpdate2 OutputSignal(4,1)
#define outPeriodUpdate3 OutputSignal(4,2)

#define outTime OutputSignal(5,0)

#define outPhaseVa OutputSignal(6,0)
#define outPhaseVb OutputSignal(6,1)
#define outPhaseVc OutputSignal(6,2)

#define outPhase1Max OutputSignal(7,0)
#define outPhase2Max OutputSignal(7,1)
#define outPhase3Max OutputSignal(7,2)
```

Constants and Global Variables A set of constants and global variables are defined for various purposes:

```
const float M_PI = 3.14159265359;
float dTheta = ((2*M_PI)/((float)SWALPHAS/2));

int threshold = 0; // Harmonics to calculate
double VPN1 = 0.0;
double VPN2 = 0.0, VPN3 = 0.0, Vdc = 0, VdcHalf = 0, lastVPN1State = 0,
lastVPN2State = 0, lastVPN3State = 0, lastVPN1 = 0, lastVPN2 = 0, lastVPN3 =
    ↪ 0;
```

```
Data Structures
electricMotor Structure
This structure holds information about the motor:

\begin{lstlisting}[language=C]
typedef struct electricMotor {
    double rotorSpeed, rotorAngle;
    int polePairs;
} electricMotor;
```

switchingTimestamp Structure This structure represents a timestamp for switching events, forming a doubly linked list:

```
typedef struct switchingTimestamp {
    float timestamp;
    double voltage;
    struct switchingTimestamp *next;
    struct switchingTimestamp *prev;
} switchingTimestamp;
phase Structure
```

This structure holds various parameters for each phase:

```
typedef struct phase {
    double voltage, prevVoltage;
    float beginTimePeriod, deltaT, periodAverage, lastPeriodAverage,
        ↪ lastRiseTimestamp,
    lastFallTimestamp, lastHalfPeriodTimestamp, lastHalfPeriodTime,
        ↪ maxVoltage, goalVoltage,
    prevGoalVoltage, lastPeriodTimestamp, phase, rotorSpeed, thetaMid;
    int state, listSize, halfBridgeState, dThetasPassed, dThetasPassedLast;
    bool modReset, modState, stateReset, stateReset2;
    float comingHarmonics[NUM_HARMONICS];
    switchingTimestamp *alphaTraverser;
} phase;
```

pulsewidthmodulation Structure This structure holds information about the PWM parameters:

```
typedef struct pulsewidthmodulation {
    double F, tolerance;
    float T, startTime, onTime, endTime;
} pulsewidthmodulation;
Function Definitions
addswitchingTimestamp Function
```

```
This function adds a new switching timestamp to the linked list:

\begin{lstlisting}[language=C]
void addswitchingTimestamp(switchingTimestamp **head, float timestamp,
    ↪ double voltage) {
    switchingTimestamp *newTimestamp = (switchingTimestamp *)malloc(sizeof(
        ↪ switchingTimestamp));
    if (newTimestamp == NULL) {
        fprintf(stderr, "Memory␣allocation␣failed\n");
        exit(EXIT_FAILURE);
    }

    newTimestamp->timestamp = timestamp;
    newTimestamp->voltage = voltage;
    newTimestamp->next = *head;
    newTimestamp->prev = NULL;

    if (*head != NULL) {
        (*head)->prev = newTimestamp;
    }

    *head = newTimestamp;
}
```

This function dynamically allocates memory for a new switchingTimestamp node, initializes it with the provided timestamp and voltage, and inserts it at the beginning of the list.

calculateHarmonics Function This function calculates the Fourier expansion for specified harmonics based on the switching events:

```
float calculateHarmonics(double Vdc, switchingTimestamp *head, float *
    ↪ harmonics, float beginTime, float rotorSpeed) {
    if(head == NULL) return -1;
    float lastTimestamp = 0; //start at ending = always a half period is
        ↪ completed
    float endTime = head->timestamp;
    float deltaT = endTime - beginTime;

    for (int m = 1; m <= NUM_HARMONICS; m++) {
        float sum = 0.0;
        switchingTimestamp *current = head;
        lastTimestamp = current->timestamp;
        current = current->next;
        if(current == NULL) return -1;
```

```
        float timestamp = current->timestamp;

        while (current != NULL) {
            timestamp = current->timestamp;
            sum += (current->voltage) / ((float)m * M_PI) * (cos((float)m* (
                ↪ timestamp-beginTime)/deltaT*2*M_PI) - cos((float)m* (
                ↪ lastTimestamp-beginTime)/deltaT*2*M_PI));
            lastTimestamp = timestamp;
            current = current->next;
        }
        harmonics[m - 1] = sum;
    }
    return deltaT;
}
```

This function traverses the linked list of switching events, computing the Fourier coefficients for each harmonic up to NUM_HARMONICS. It uses the difference in timestamps to calculate the harmonic content in the voltage signal.

calculateComingHarmonics Function This function computes the harmonics for the upcoming period:

```
float calculateComingHarmonics(switchingTimestamp *head, float *harmonics) {
    if(head == NULL) return -1;
    float lastTimestamp = head->timestamp;
    float endTime = head->timestamp;
    switchingTimestamp *current = head->next;
    if(current == NULL) return -1;
    float deltaT = (current->timestamp - endTime);
    for (int m = 1; m <= NUM_HARMONICS; m++) {
        float sum = 0.0;
        while (current != NULL) {
            float timestamp = current->timestamp;
            sum += current->voltage / (m * M_PI) * (cos(m * timestamp /
                ↪ deltaT * 2 * M_PI) - cos(m * lastTimestamp / deltaT * 2 *
                ↪ M_PI));
            lastTimestamp = timestamp;
            current = current->next;
        }
        harmonics[m - 1] = sum;
    }
    return 1;
}
```

Similar to calculateHarmonics, this function calculates the upcoming harmonics using the same principles but focused on the next period of the waveform.

calculateThetaK Function This function computes the mid-angle thetaK for a given voltage and switching angles:

```
float calculateThetaK(float thetaLo, float thetaHi, float maxVoltage, float
    ↪ Vdc) {
    return thetaHi/2 + thetaLo/2 + (maxVoltage * (cos(thetaHi) - cos(thetaLo)
        ↪ )) / Vdc;
}
```

This function computes the thetaK, which is the mid-point between thetaLo and thetaHi, adjusted for the desired maxVoltage.

addHarmonicCompensation Function This function adjusts thetaK to compensate for specific harmonics:

```
float addHarmonicCompensation(float thetaK, float thetaLo, float thetaHi,
    ↪ float h, float hc, float Vdc) {
    thetaK += (hc * (cos(h * thetaLo) - cos(h * thetaHi))) / (h * Vdc);
    return thetaK;
}
```

This function adds a compensation term to thetaK, based on harmonic content, improving the waveform's harmonic profile.

calculateComingPeriod Function This function determines the PWM sequence for the upcoming period, based on harmonic compensation:

```
int calculateComingPeriod(switchingTimestamp** head, switchingTimestamp**
    ↪ traverser, float maxVoltage, float Vdc, float comingHarmonics[], bool
    ↪ alternateMode) {
    float dTheta = 2 * M_PI / SWALPHAS;
    float prevTheta = 0;
    float theta = 0;
    float thetaLo = 0;
    float thetaHi = dTheta;
    float thetaK = 0;
    int maxListSize = (int)((float)SWALPHAS/2.0);
    int listSize = 0;
    if(*head == NULL) return -1;

    emptyList(head);

    for(int i = 0; i < maxListSize; i++) {
        if(alternateMode) {
            theta = dTheta * (float)(i+1);
```

```
        } else {
            theta = dTheta * (float)i;
        }
        thetaLo = prevTheta;
        thetaHi = theta;

        if(i == 0) {
            addswitchingTimestamp(head, 0, maxVoltage);
            prevTheta = thetaHi;
            listSize++;
            continue;
        }

        thetaK = calculateThetaK(thetaLo, thetaHi, maxVoltage, Vdc);

        for(int h = 1; h < NUM_HARMONICS; h++) {
            thetaK = addHarmonicCompensation(thetaK, thetaLo, thetaHi, (float
                ↪ )h, comingHarmonics[h], Vdc);
        }

        if(i == (maxListSize-1)) {
            addswitchingTimestamp(head, 1, -maxVoltage);
            prevTheta = thetaHi;
            listSize++;
            continue;
        }

        addswitchingTimestamp(head, thetaK / (2 * M_PI), -maxVoltage);
        prevTheta = thetaHi;
        listSize++;
    }

    *traverser = *head;
    return 1;
}
```

This function calculates the switching times for the upcoming period, taking into account harmonic compensation to minimize unwanted harmonics in the signal. It dynamically constructs the switching timestamps list for the PWM controller to follow.

emptyList Function This function frees the memory of the linked list and resets the head pointer:

```
void emptyList(switchingTimestamp** head_ref) {
```

```
    switchingTimestamp* current = *head_ref;
    switchingTimestamp* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *head_ref = NULL;
}
```

This function iterates through the linked list, freeing each node's memory, and sets the head pointer to NULL to signify the list is empty.

outputHarmonicConstants Function This function outputs harmonic constants for specified phases:

```
int outputHarmonicConstants(int num, double *harmonics) {
    for (int i = 0; i < NUM_HARMONICS; i++) {
        if(num == 1) {
            *outV1HC1 = harmonics[0];
            ...
            *outV1HC13 = harmonics[12];
        } else if(num == 2) {
            *outV2HC1 = harmonics[0];
            ...
            *outV2HC13 = harmonics[12];
        } else if(num == 3) {
            *outV3HC1 = harmonics[0];
            ...
            *outV3HC13 = harmonics[12];
        }
    }
    return 1;
}
```

This function sets the output signals for harmonic constants, based on the phase number (num). It updates the output signals with the computed harmonic values for each phase.

Utility Functions Several utility functions are defined for common mathematical operations:

```
double signD(double num) {
    return (num >= 0.0) ? 1.0 : -1.0;
}
```

```
int signI(int num) {
    return (num >= 0) ? 1 : -1;
}

float signF(float num) {
    return (num >= 0.0f) ? 1.0f : -1.0f;
}

float max(float num1, float num2) {
    return (num1 > num2) ? num1 : num2;
}

float min(float num1, float num2) {
    return (num1 < num2) ? num1 : num2;
}

float constrain(float num, float lo, float hi) {
    if (num < lo) {
        return lo;
    } else if (num > hi) {
        return hi;
    } else {
        return num;
    }
}

float avg(float num1, float num2) {
    return (num1 + num2) / 2.0;
}
```

signD, signI, signF: Return the sign of a number (double, int, float respectively). max: Return the maximum of two float numbers. min: Return the minimum of two float numbers. constrain: Constrain a float number to lie between two specified bounds. avg: Calculate the average of two float numbers. Conclusion The code implements a sophisticated motor control system with a focus on PWM and harmonic compensation. It uses structures and linked lists to manage and optimize the motor's performance. The provided functions calculate and adjust switching times to minimize harmonic distortion, ensuring efficient and smooth motor operation. Each function and variable plays a crucial role in the overall system, contributing to the accurate simulation and control of the motor's behavior.

G.0.2 Output code explanation

This report delves into a motor control system algorithm, focusing on the continuous loop execution for managing the voltage, phase, and modulation states of a motor's rotor. The algorithm intricately handles real-time updates and calculations to ensure the efficient operation of the motor, based on various input parameters such as rotor speed, rotor angle, and voltage goals.

Initialization The initial phase of the code involves setting up the essential variables required for the motor's operation:

```
Vdc = inVdc;
VdcHalf = Vdc / 2;
threshold = inThreshold;
time = (float)inTime;

outTime = time;
motor.rotorSpeed = inRotorSpeed;
motor.rotorAngle = inRotorAngle;
```

Here, the input voltage (Vdc), threshold values and time parameters are initialized. The motor's rotor speed and angle are also set based on the input values.

Phase 1 Calculations Voltage and Goal Voltage Update In the first phase, the system updates the voltage and goal voltage for phase 1:

```
phase1.prevVoltage = phase1.voltage;
phase1.voltage = inVPN1;

phase1.prevGoalVoltage = phase1.goalVoltage;
phase1.goalVoltage = inVaGoal;
```

The previous voltage and goal voltage are stored before updating them with new input values.

Half-Bridge State Determination The half-bridge state is determined based on the voltage threshold:

```
if (phase1.voltage >= threshold) {
  phase1.halfBridgeState = 1;
} else {
  phase1.halfBridgeState = -1;
}
```

This condition sets the state to 1 if the voltage is above the threshold and to -1 otherwise.

Phase Calculation The phase of the motor's rotor is calculated if the rotor speed is non-zero and the list size is greater than zero:

```
if (fabs(phase1.rotorSpeed) > 0 && phase1.listSize > 0) {
```

```
  phase1.phase = (time - phase1.beginTimePeriod) / (2 * M_PI / ((float)
      ↪ phase1.rotorSpeed * (float)motor.polePairs)) * 2 * M_PI;
  outPhaseVa = phase1.phase;

  phase1.dThetasPassed = (int)(phase1.phase / dTheta);

}
```

The phase calculation involves complex mathematical operations considering the time, rotor speed, and pole pairs.

State Reset and Modulation The code further handles state resets, based on specific phase conditions:

```
if ((phase1.phase > (float)0.5 * M_PI && phase1.stateReset)) {
  phase1.state = signF(phase1.goalVoltage);
  phase1.stateReset = false;
}
if (phase1.phase > (float)1.5 * M_PI && phase1.stateReset2) {
  phase1.state = signF(phase1.goalVoltage);
  phase1.stateReset2 = false;
}
```

These conditions reset the state and manage modulation accordingly.

Voltage and Harmonics Calculations The algorithm updates voltage on rising and falling edges of the square wave and calculates the harmonics:

```
if ((phase1.voltage > threshold && phase1.prevVoltage <= threshold && (time
    ↪ - phase1.lastRiseTimestamp) > pwm.T * pwm.tolerance && phase1.modState
    ↪  == false)) {
  phase1.modState = true;
  phase1.lastRiseTimestamp = time;
  addswitchingTimestamp(&switchingTimestampsVPN1, time, Vdc / 2);
  phase1.listSize++;
  ...
} else if ((phase1.voltage < threshold && phase1.prevVoltage >= threshold &&
    ↪  (time - phase1.lastFallTimestamp) > pwm.T * pwm.tolerance && phase1.
    ↪ modState == true)) {
  phase1.modState = false;
  phase1.lastFallTimestamp = time;
  addswitchingTimestamp(&switchingTimestampsVPN1, time, -Vdc / 2);
  phase1.listSize++;
}
```

Switching timestamps are added on detecting voltage changes, and the list size is updated. Harmonics are then calculated based on these timestamps.

Phase 2 Calculations The second phase follows a similar structure as phase 1, handling voltage updates, state determination, phase calculations, and modulation for the second set of inputs:

```
phase2.prevVoltage = phase2.voltage;
phase2.voltage = inVPN2;

phase2.prevGoalVoltage = phase2.goalVoltage;
phase2.goalVoltage = inVbGoal;

if (phase2.voltage >= threshold) {
  phase2.halfBridgeState = 1;
} else {
  phase2.halfBridgeState = -1;
}

if (fabs(phase2.rotorSpeed) > 0 && phase2.listSize > 0) {
  phase2.phase = (time - phase2.beginTimePeriod) / (2 * M_PI / ((float)
      ↪ phase2.rotorSpeed * (float)motor.polePairs)) * 2 * M_PI;
  outPhaseVb = phase2.phase;

  phase2.dThetasPassed = (int)(phase2.phase / dTheta);

}
```

The operations performed in this phase mirror those in the first phase, ensuring consistent control across different phases of the motor.

Conclusion The continuous loop code for this motor control system demonstrates a robust mechanism for real-time voltage, phase, and modulation management. By leveraging threshold-based state determination, intricate phase calculations, and harmonic adjustments, the system ensures precise control over the motor's rotor dynamics. This detailed understanding of the code's functioning provides insights into its operational efficiency and reliability in various motor control applications.