NTNU
Norwegian University of Science and Technology

SDFT based PMU prototype

Isidora Radevic

Wind Energy
Submission date: September 2018
Supervisor: Olav B Fosso, IEL

Norwegian University of Science and Technology
Department of Electric Power Engineering

SDFT based PMU prototype

by

Isidora Radević

to obtain the degree of Master of Science in Electrical Engineering
at the Delft University of Technology,
to be defended publicly on Wednesday July 18, 2018 at 14:00.

Student number:	4615905	
Project duration:	December 1, 2017 – July 18, 2018	
Thesis committee:	Dr. ir. M. Popov,	TU Delft, supervisor
	Prof. dr. P. Palensky,	TU Delft, professor
	Prof. dr. O. Fosso,	NTNU, co-supervisor
	Dr. ir. O. Mansour,	Smart State Technology, company supervisor
	Dr.ir. D.J.A. Bijwaard,	Smart State Technology, company supervisor

TU Delft

Contents

List of Figures

1 Introduction

Electrical power system is going through the period of significant transformation, due to the increasing role of renewable and decentralized energy sources. There is a need for improved reliability and security of energy production, transmission and distribution, especially in power networks with a high level of operational uncertainties, in order to reduce the number of catastrophic blackouts. In the past, it was common that the power grids are operated in a load-driven mode, meaning that the load is statistically predictable and generation scheduled accordingly. The caused deviations in power balance are then compensated in run-time. However, this approach is possible only if the generation is fully controllable, what it is hardly achievable due to the intermittent nature of the power generation. Consequently, the power systems are becoming generation-driven, where a generation leads and the rest of the system adjusts. In order to maintain power stability in such system, new electrical (e.g. advanced wide-area control, protection and monitoring solutions (WAMPAC)), chemical (e.g. batteries and hydrogen) and thermal technologies (e.g. combined heat and power stations) have to be deployed. The focus of this thesis is in electrical solutions, particularly in advanced synchronized measurement technologies (SMT), that is important element and enabler of WAMPAC systems. Currently, phasor measurement units (PMUs) are the most advanced time-synchronized technology in small number present mostly in transmission systems. The advantage of PMUs over traditional digital measurement units is that besides rms value of the measured electrical quantity (current or voltage), PMUs provide the information about the phase and hence the frequency of the input signal. Moreover, the measurements are accompanied with the time-quality information (the timestamp) defining the instant of time the measurement is valid for. The phase is estimated with the reference to a global time reference, which is selected to be reliant on Global Positioning System (GPS), that gives a reliable information about the time everywhere. As a result of the synchronized measurements, with the multiple PMUs covering one area of the power grid, the detailed snapshot of the system can be obtained 10, 25 or 50 times per second. It increases the observability of the grid and in that way, proper control and protection actions can be taken in a timely manner and avoid that the disturbances cause the cascade of the system, separation of the grid into unplanned islands or in the worst scenario the complete collapse of the system. The significant improvements can be achieved in many power system areas, such as [1]:

- real-time visualization of the power system
- design of an advanced early warning system
- post mortem analysis of the causes of the system blackouts
- validation of the system models
- enhanced performance of the state estimation technique
- the congestion management in real-time
- real time voltage and frequency stability
- improvements in damping of inter-area oscillations
- design of adaptive protection and control systems

1.1. Motivation and objectives of the thesis

There are a few companies operating in the domain of electrical power, electrical equipment and automation technology areas, that are providing their solutions for advanced synchronized measurement technologies, particularly phasor measurement units. All solutions have something similar, that is the algorithm for synchrophasor estimation and the hardware platform that runs the algorithm. That is the triggering point for all researchers that tried to make they own PMU prototype by using optimized algorithms that can be supported by the low-cost hardware. By searching for some of the realized projects, interesting examples are found. The first one is the development of the PMU prototype based on the i-IpDFT (iterative - interpolated Discrete Fourier Transform) technique in the FPGA (Field Programmable Gate Arrays) hardware platform, realised in EPFL University in Lausanne [2]. The second example is OpenPMU project that provides the tools to help researchers to create new synchronised measurement technologies. The project was originally founded at Queen's University Belfast, UK, and subsequently joined by the SmarTS Lab at KTH Stockholm, Sweden [3]. The idea behind this thesis was that the department for Inteligent Electrical Power Grids (IEPG) of TU Delft gives its own contribution to the advanced synchronized monitoring solutions.

The objectives of the thesis are:

- Verification of the performance of existing, computationally efficient, Smart Discrete Fourier Transform (SDFT) algorithm for synchrophasor estimation proposed in the literature.

- Implementation of the algorithm into the embedded platform. The algorithm adaptation with respect to the hardware capacity and limitations.

- Testing the performance of the PMU prototype with respect to the requirements of IEEE C37.118 standard for Synchrophasor Measurements in Power Systems.

1.2. Organization of The Thesis

The introduction about the mathematical definitions behind the synchrophasor estimation techniques are given in the first part of the thesis report. The focus is given to the definition of the synchrophasor and the theory about Discrete Fourier Transform. Following that, the limitations and performance of the DFT Technique are highlighted. The Literature Review concludes the Chapter - Theory and Definitions, by giving the overview and comparison of different synchrophasor estimation methods and stating the reason why it is decided that the SDFT algorithm is the best candidate for the real-time implementation. The following Chapter - Phasor Measurement Units introduces the structure of the PMU device by explaining what are the building blocks that form one synchronized measurement devise. The PMU performance requirements set by IEEE C37.118 are presented in Chapter 4. The Chapter 5 explains the mathematical derivation of the SDFT algorithm and verification of the algorithm under different test conditions. The embedded system architecture is presented in Chapter 6 - LVSensors - low voltage and current measurement sensors. Finally, the taken steps and faced challenges during the real-time implementation of the algorithm in the voltage embedded sensor as well as the performance of the PMU prototype are presented in the Chapter 7. The developed Python codes are attached in the Appendix.

1.3. Methodology

The verification of the algorithm is done in the MATLAB enviroment. First of all, the SDFT algorithm is transferred from the raw mathematical equations in the paper to the code in MATLAB. Following that, the test scripts are written in order to check the performance of the algorithm for different testing conditions modelled by changing the parameters of the input signal. The real-time implementation of the algorithm is performed by using Python programming language. It was a challenging task since the data coming from the A/D converted should be processed through the algorithm as soon as they arrive. Otherwise, the data start to accumulate in the query and the implementation is not in the real-time anymore. The communication module of the PMU prototype is realized by using pyPMU module that formats the measurement output stream according to the IEEE standard for Data Transfer. The testing of the PMU prototype is realized by using the Real Time Digital Simulator(RTDC), RSCAD software interfacing the RTDS and Omicron amplifier.

1.4. Contribution of the Thesis

The main contribution of the thesis is the development of the SDFT based Phasor Measurement Unit - a synchronized device that accurately estimates the power grid values (voltages, currents and frequency) in real-time. The main focus of the thesis is in finding the way how to obtain the best performance of the PMU prototype, by optimizing SDFT based computationally efficient algorithm for synchrophasor estimation to the low-cost embedded hardware platform. Since, the present PMU solutions are not massively deployed in the grid due to the high expenses, the thesis contributes in providing a low-cost solution with the high estimation accuracy. Large-scale integration of SDFT based PMUs into the power system would significantly increase observability of the distribution grid and enable more efficient control and protection actions.

2
Theory and Definitions

2.1. Signal Model

In traditional electrical power systems, during normal operating and steady state conditions, current and voltage waveforms are usually modelled as sinusoidal signals with constant parameters:

$$x(t) = A_0 * cos(2\pi f_0 t + \phi_0) \tag{2.1}$$

where A_0 represents the amplitude, ϕ_0 phase, and f_0 frequency of the signal. However, electrical power grid is rarely in steady state and signal parameters vary with the time. The most evident dynamic state condition is frequency variation from its nominal value, due to the generation/load imbalance. Similarly, due to the transient phenomena, waveform signal and phase can be also affected. There are different types of transients that appear in the grid and they can be classified as following:

1. Harmonics, namely components of the signal at frequency that is integer multiple of the fundamental frequency. These components are usually generated by power electronic devices in transmission systems, such as Flexible AC Transmission Systems or HVDC connections, or by converters connected to distributed generation units in distribution grids.
2. Inter-harmonics, components with frequencies between two consecutive harmonics or those components whose frequencies are not integer multiples of the fundamental power frequency.
3. Sub-harmonics, a special subset of inter-harmonics that have frequency values that are less than that of the fundamental frequency.
4. aperiodic components with DC offset, that usually appear during the disturbances in the grid caused by short-circuit faults or inrush currents of the transformers and induction motors.
5. noise, added by the measurement equipment and the grid itself.

Consequently, the model of the AC grid signal shown in equation 2.1 is not general representation that covers all expected signals coming from the grid. Hence, the formula can be expanded in following way[4]:

$$x(t) = A(t) * cos(2\pi f(t)t + \phi_0) + \sum_{h=1}^{H} cos(2\pi h * f_0(t)t + \phi_h) + A_{DC} * e^{-t/\tau} + \epsilon_n \tag{2.2}$$

where, the first term represents the dominant tone of the spectrum, the second term sums inter- and sub-harmonics, the third term includes aperiodic DC component and the last term adds the white noise.

2.2. Definition of the Phasor, Frequency and ROCOF

Signal $x(t)$ in Equation 2.1 can be represented as a phasor X with magnitude $X/\sqrt{2}$ with initial phase ϕ_0 rotating in the complex plane with angular frequency $\omega = 2\pi f$ in radians/s, shown in figure 2.1:

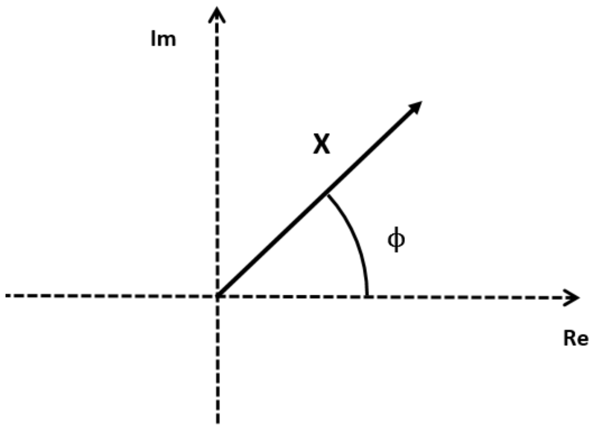


Figure 2.1: Phasor representation in complex plain

Equation 2.1 can be written as [5]:

$$x(t) = Re(A_0 e^{(\omega t+\phi_0)}) = Re(e^{j\omega t} A_0 e^{j\phi_0}) \tag{2.3}$$

Now, the phasor representation can be extracted as:

$$X = (A_0/\sqrt{2})e^{j\phi_0} \tag{2.4}$$

where $A_0/\sqrt{2}$ is the root mean square (RMS) value of the input signal.
The phasor presentation is only possible for pure sinusoidal signals. So, in cases when the signal is distorted with other signals at different frequencies, it becomes necessary to extract main tone spectral component and then represent it by a phasor. It can be obtained by a technique called Fourier transform, explained in the section 2.4. Besides a phasor, there are two important parameters more that indicate the grid behaviour: frequency and rate of change of frequency(ROCOF). Frequency can be represented as rate of change of angle:

$$f = \frac{\partial \psi}{\partial t} \tag{2.5}$$

where $\psi = 2\pi f_0 t + \phi_0$, whereas, ROFOC can be calculated as:

$$ROCOF = \frac{\partial^2 \psi}{\partial^2 t} = \frac{\partial f}{\partial t} \tag{2.6}$$

2.3. Definition of the Synchrophasor

As stated in previous section, the phasor representation is valid only in sinusoidal steady state conditions. However, even when the power system is in dynamic state, the variations in state variables are slow and it can be treated as series of steady state conditions (quasi-steady state), so the phasor representation can be still in use. But this time, in order to compare the phasors obtained from remote sites, it is necessary that the phasors are referenced to a common time reference. For this reason, the term phasor is extended to the synchrophasor defined in the IEEE Standard for Synchrophasor Measurements [6] as:
[...] The *synchrophasor* representation of the signal x(t) in is the value X where ϕ is the instantaneous phase angle relative to a cosine function at the nominal system frequency synchronized to UTC (Coordinated Universal Time) [...]

2.4. Discrete Fourier Series and Transform

2.4.1. Fourier series representation of discrete-time periodic signals

The Fourier series theory states that a discrete-time periodic signal can be represented as a linear combination of harmonically related complex exponentials. Consider a discrete-time periodic signal x[n] with the period N:

$$x[n] = x[n+N] \tag{2.7}$$

The Fourier series representation of the discrete-time signal is:

$$x[n] = \sum_{k=1}^{N-1} a_k e^{jk(2\pi/N)n} \tag{2.8}$$

where:

- $\frac{2\pi}{N}$ is the frequency of the discrete-time signal
- a_k are referred to as Fourier series coefficients, the weighing coefficients for different frequency components existing in the discrete-time signal.
- k is a factor taking values from 0 to N-1 only. The reason for a limited range is that exponential term $e^{jk(2\pi/N)n}$ repeats after N successive values of k.

After few steps of derivation, a_k can be extracted and calculated as:

$$a_k = \frac{1}{N}\sum_{n=1}^{N-1} x[n]e^{-jk(2\pi/N)n} \tag{2.9}$$

Since, the PMUs report the phasor of a main tone of the spectrum, only one value of k will be considered (k=1, if only one cycle of the signal is used for a phasor estimation).

2.4.2. Fourier transform representation of discrete-time periodic signals

Fourier series coefficients a_k represent the samples of an envelope function. As number N increases, samples are more closely spaced. If N approaches infinity, samples are forming envelope function following expression:

$$X(e^{-j\omega}) = \sum_{n=-\infty}^{+\infty} x[n]e^{-j\omega} \tag{2.10}$$

$X(e^{-j\omega})$ is referred to as Discrete-time Fourier transform.
Coefficients a_k are proportional to samples of $X(e^{-j\omega})$:

$$a_k = \sum_{n=-\infty}^{+\infty} x[n]e^{-jk\omega_0} = \frac{1}{N}X(e^{-j\omega_0}) \tag{2.11}$$

where $w_0 = \frac{2\pi}{N}$ is spacing of the samples in the frequency domain. Now, the input signal x[n] can be represented as:

$$x[n] = \sum_{k=1}^{N} \frac{1}{N}X(e^{-j\omega_0})e^{-jk\omega_0 n} \tag{2.12}$$

As $\omega_0 = 2\pi/N$, the function can be further modified to:

$$x[n] = \frac{1}{2\pi}\sum_{k=1}^{N} X(e^{-j\omega_0})e^{-jk\omega_0 n}\omega_0 \tag{2.13}$$

As N increases, ω_0 decreases, and as N$\rightarrow\infty$, equation 2.13 passes to an integral:

$$x[n] = \frac{1}{2\pi}\int_{2\pi} X(e^{-j\omega_0})e^{-j\omega n}d\omega \tag{2.14}$$

Equation 2.10 as $synthesis$ equation and equation 2.14 as $analysis$ equation are referred to as discrete-time Fourier transform pair [7].

2.5. Performance and Limitations of DFT Technique

Discrete-time Fourier transform is a powerful technique for signal spectrum analysis. However, in order to use DFT for AC signals in power systems, we should be aware of some limitations of the technique. Following two sections explain DFT accuracy for phasor estimation in case of nominal and off-nominal state conditions.

2.5.1. Phasor Estimation of the Input at Nominal Frequency

Since only fundamental frequency component will be taken in consideration, in following sections it will be assumed that factor k takes value 1. After calculating Fourier series coefficients, the phasor X_k can be estimated as:

$$X_k = \frac{1}{\sqrt{2}}(2 * Re(a_k) + j2 * Im(a_k)) = \sqrt{2}(Re(a_k) + jIm(a_k)) \tag{2.15}$$

Considering that the phasor estimation is a continuous process, it is necessary to deploy an algorithm that will update the phasor estimation as new data samples are obtained. There are two methods: recursive and non-recursive. Both of them are relaying on a sliding window N samples long that moves through the signal by one sample step. However, non-recursive method repeats DFT calculation in each window, whereas recursive method only updates the phasor estimated in previous window. By using non-recursive method, two consecutive phasors X^{N-1} and X^N can be obtained in following way [5]:

$$X^{N-1} = \frac{\sqrt{2}}{N}\sum_{n=0}^{N-1} x_n e^{-jn\theta} \tag{2.16}$$

$$X^{N} = \frac{\sqrt{2}}{N}\sum_{n=0}^{N-1} x_{n+1} e^{-jn\theta} \tag{2.17}$$

where $\theta = \frac{2\pi}{N}$ and x_n part of the input signal x[n] considered by the window. As a consequence of this approach, the estimated phasor will have constant magnitude and will rotate by angle θ as the data window advances by one sample (assuming that the input signal amplitude does not vary). However, produced angle degradation can be easily suppressed. Non-recursive methods are numericaly stable, but they require redundant computation effort, since the DFT is repeated for each window. On the other hand, recursive method is computationally efficient, but can experience numerical problems. If r is the first sample in data window, and hence r+N the last sample, the recursive phasor estimate is given by:

$$X^{N+r} = X^{N+r-1} + \frac{\sqrt{2}}{N}(x_{N+r} - x_r)e^{-jr\theta} \tag{2.18}$$

In case if there is no difference between two consecutive phasors, the update $\frac{\sqrt{2}}{N}(x_{N+r} - x_r)e^{-jr\theta}$ is zero. Otherwise, the update is added to previous value. However, the disadvantage of this method appears if the previously estimated phasors are erroneous, because the estimation error accumulates and leads to numerically unstable estimation.

Based on phasor estimation, frequency and ROCOF can be obtained by using equations 2.5 and 2.6. So, any error accumulated in estimated phasor will directly influence accuracy of frequency and ROCOF estimates.

2.5.2. Phasor Estimation of the Input at Off-Nominal Frequency

By assuming that the input signal at 50 Hz is sampled at 2kHz, one rectangular window covering one full signal cycle, consists of 40 samples. The discrete-time signal is shown in blue color in figure 2.2. Now, let's consider the same input signal but at frequency of 48 Hz (in red). This time the same window length is not enough to cover complete cycle of the signal. Consequently, the DFT estimated phasors calculated for the samples in the window will not be accurate.

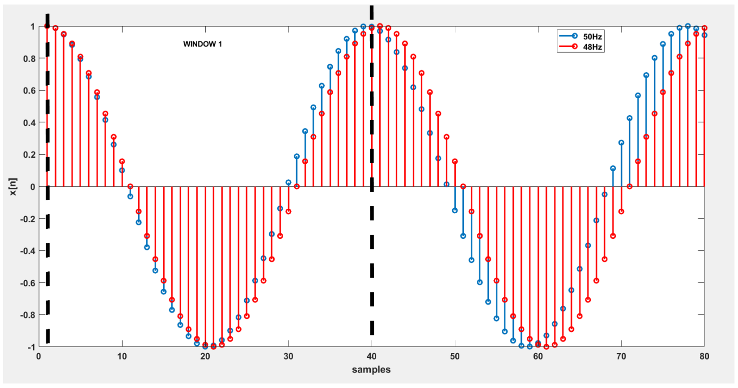


Figure 2.2: Two signals at different frequencies

In other words, the process of windowing consists of multiplying the infinite sequence of samples $x(n)$ by the rectangular function $w_r(t)$. If the input signal $x(t)$ is sinusoidal wave at nominal frequency of 50 Hz (equation 2.19), its corresponding Fourier Transform is $X(f)$ (equation 2.21):

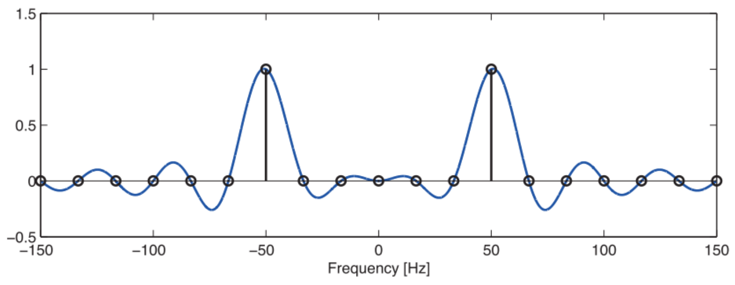
$$x(t) = Acos(2\pi f_0 t) \tag{2.19}$$

$$X(f) = \frac{A}{2}[\delta(f - f_0) + \delta(f + f_0)] \tag{2.20}$$

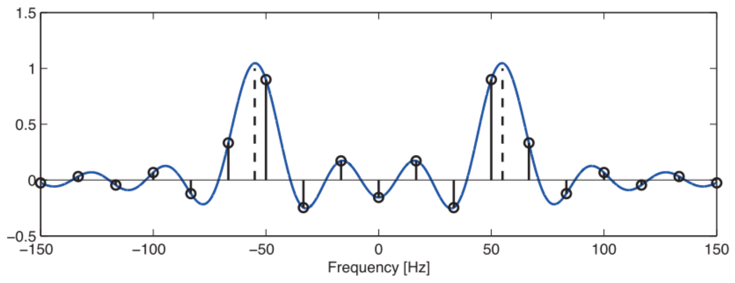
Since the multiplication in time domain is equivalent to convolution in frequency domain, the windowing operation can be modeled as:

$$x_{s,w}(t) = w_r(t) \cdot x_s(t) = w_r(t) \cdot [x(t) \cdot s(t)] = [w_r(t) \cdot x(t)] \cdot s(t) \underset{\longrightarrow}{\Im} [W_r(f) * X(f)] * S(f) = X_{s,w}(f) \tag{2.21}$$

where $x_s(t)$ is sampled sinusoidal input function, and s(t)=$\sum_{n=-\infty}^{\infty} \delta(t - nTs)$ So, the spectrum of the function $x_{s,w}$ ($X_{s,w}$) is composed of Fourier transform of sampled input signal convolved by Fourier transform of windowing function ($X_{s,w}$) shifted by multiples of the sampling rate $\frac{1}{Ts}$ and then superimposed. Since the windowing function is rectangular $W_r(f)$ is a sinc function $sinc(fT) = \frac{sin(\pi f t)}{\pi f t}$, DFT spectrum for the inputs signal at nominal frequency and spectrum under off-nominal frequency conditions are shown in figure 2.3



(a)



(b)

Figure 2.3: DFT spectrum under nominal and off-nominal frequencies [8]

During the nominal conditions, the frequency spectrum consists of two sinc functions centered around $+/-f_0$, having the non-zero value only at nominal frequencies. However, in case of the conditions when the frequency vary from the nominal value, bins at other frequencies experience a non-zero values. This is called spectral leakage effect that deteriorate accuracy of the DFT algorithm. Additionally, main lobe and side lobes of the windowing function contributes reduced accuracy of the DFT method. Namely, the main lobe width can cause difficulties in identifying main tone of the spectrum, while side lobes can generate so-called spectral interference between nearby tones. Other windowing functions are recommended in order to enhance the DFT accuracy. One of them is Hanning, shown in figure 2.5. As it can be seen, compared to rectangular window (2.4), Hanning window has attenuated side lobes.

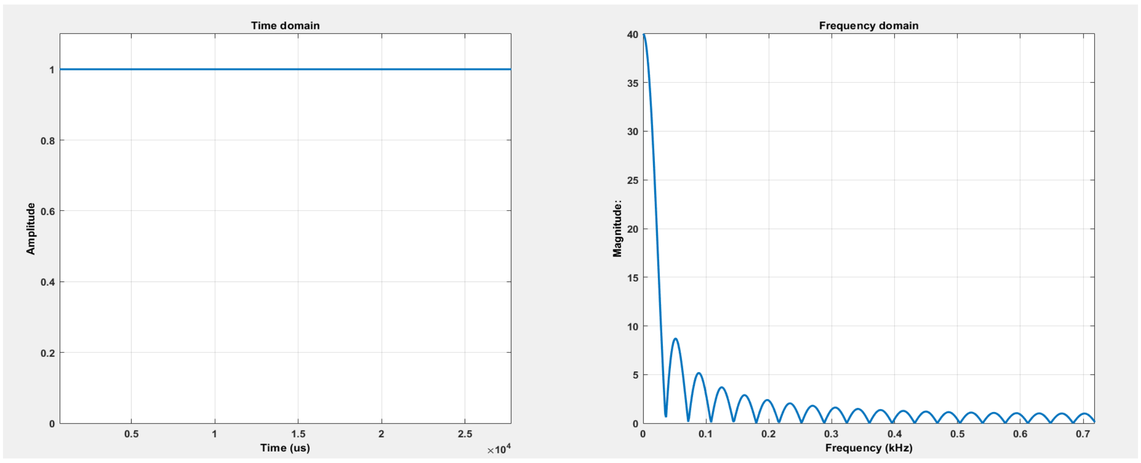


Figure 2.4: Rectangular window

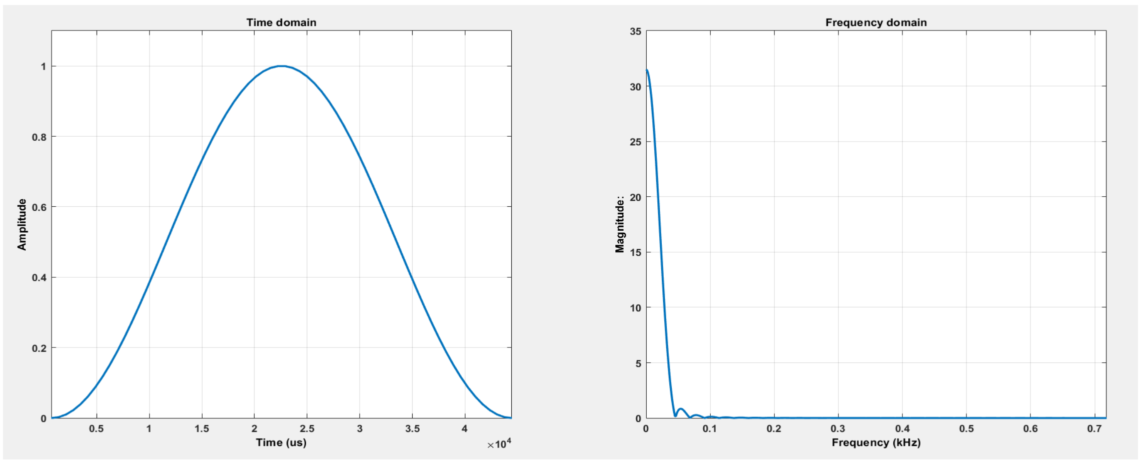


Figure 2.5: Hanning window

2.6. Literature Review

As explained in the previous section, off-nominal grid conditions can significantly influence accuracy of phasor estimation technique. In order to minimize an error, different methods are recommended in the literature, such as:

- variable window length (figure 2.6), namely by adapting the window length to the frequency of the signal, the leakage effect can be minimized. However, since the sampling period is fixed, the possible fundamental frequencies are discrete and in the most of the cases small leakage effect will remain [9].

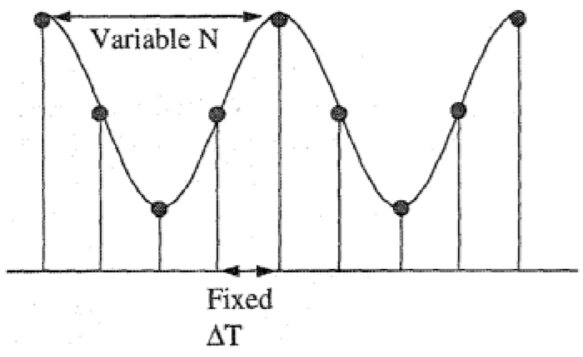


Figure 2.6: Variable window size technique [9]

- variable sampling rate (figure 2.7), namely by adapting the sampling rate to be the integer multiple of the signal frequency, it can be achieved that fixed window always captures complete cycle. Only disadvantage of this technique are stability issues in the hardware part, since adjustments are done in A/D converter by using advanced control loops [10].

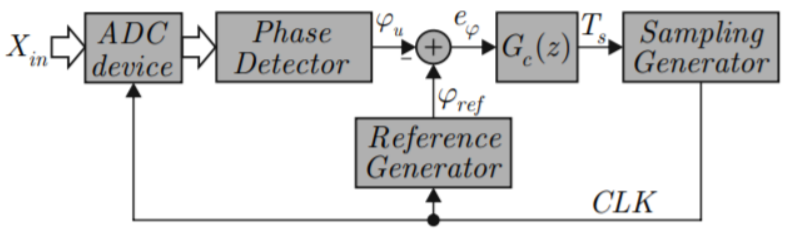


Figure 2.7: The technique of variable sampling frequency [10]

- Interpolated-iterative DFT technique, based on an interpolation method for finding the correction term δ that better approximates the exact location of the main spectrum tone [8]. The drawback of the such approach is that A/D converter with high sampling rate is required, as well as high performance processor to deal with the huge amount of data and run demanding iterative algorithm.

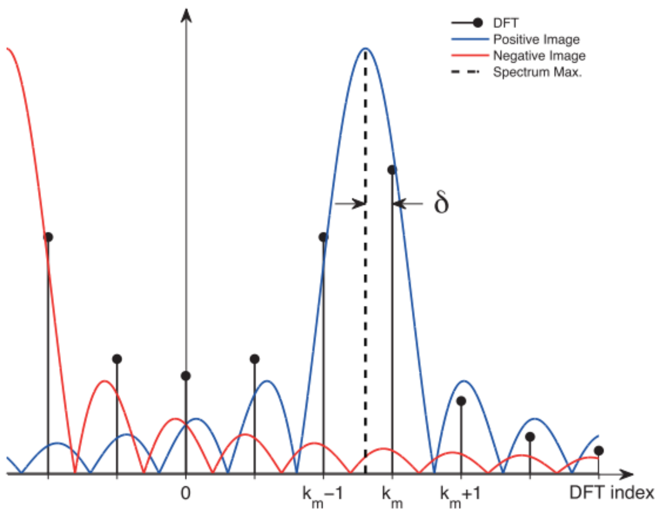


Figure 2.8: Interpolated-iterative DFT technique for finding a correction δ [8]

- Smart Discrete Fourier Transform (SDFT) technique, that relies on accurately estimated system frequency in order to estimate the magnitude and the phase of the input signal [11]. The SDFT algorithm appears to be a better alternative for the commonly used standard DFT algorithm to estimate phasors in the presence of off-nominal frequencies, noise and harmonics. The algorithm requires additional computational steps compared to the standard DFT, but the benefits of reduced filtering requirements can compensate the additional computational requirements. This algorithm is further extended for synchrophasor estimation and implemented in the PMU model of the Real-Time Digital Simulator [12]. For stated reasons, SDFT technique is chosen, further modified and deployed in PMU prototype. It is explained more in details in chapter 7.

3
Phasor Measurement Units

3.1. Architecture of PMU

In this chapter, the architecture of PMU will be discussed in details. The block scheme of the main components of PMU is illustrated in figure 3.1. Specific information about the purpose of each component and its connection with others is given in following sections.

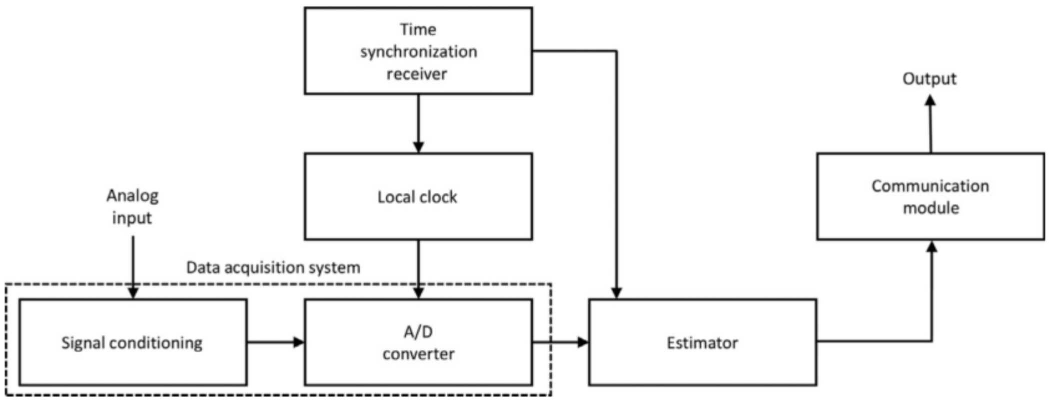


Figure 3.1: Architecture of PMU [13]

3.1.1. Data Acquisition Module

The Data Acquisition Module (DAQ) is the front-end of the Phasor Measurement Unit. It consists of the analog signal conditioning and the analogue-to-digital converter (ADC). The signal conditioning considers the adapting the voltage and current values to the digital acquisition circuits by the Voltage and Current Transformers [14]. The most important purpose of the conditioning system for synchrophasor measurements is to avoid significant modifications of the signal around the fundamental system frequency. That can be achieved by low pass anti-aliasing filter intended to reduce the band of the electrical signals before analog-to-digital conversion. Its cut-off frequency depends on what is the relevant information for the computational module. In a basic PMU approach, it is around the fundamental system frequency. Attenuation or phase distortion have direct impact on the measurement, while delays can affect synchronization of the measurements and measurement reporting latency. Flat response of the low-pass filter in the pass-band around 50 Hz and attenuated phase distortion are necessary to reduce phasor estimation errors. The ADC produces the digital signal to be elaborated by the computational unit that runs the estimation algorithm. Its sampling frequency is precisely connected to the anti-aliasing filter frequency response and its input range is set by considering the voltage or frequency range considered for input signals. There are at least six channels, corresponding to the three-phase voltages and currents.
It is important to understand how the acquired samples can be accurately referred to their time reference, so

that the following phasor estimation can be time-aligned to the reporting instants and correctly time-tagged. The following section will explain how the synchronization is achieved and what are the available synchronization sources.

3.1.2. Synchronization Module

A clock is a system of circuits consisting of two main parts: an oscillating device providing a reference time interval and a counter device that by counting the intervals indicates time. There are different types of oscillators such as mechanical, as pendulum or balance wheel, or electronic, as Quartz-crystal oscillators. The electronic oscillators are today the most widely used due to the low price and low power consumption. The best stability and accuracy can be obtained with atomic clocks, that use an electron transition frequency in the microwave, optical, or ultraviolet region of the electromagnetic spectrum of atoms as a frequency standard for its time-indicating element. [15] Counters are structured in a cascade array that sums up the accurate oscillations from the reference device. Seconds, minutes and hours are generated in the way that each level is triggered by different number of oscillations. Due to environmental changes such as a temperature or mechanical stress the oscillation frequency of the reference device can be affected, that will cause a limited stability. As a consequence, periodic clock synchronization is required to keep the time within a certain time offset with respect to a reference time. There are different time-scales that are in use in different fields. The most utilized and based on the measurements produced by atomic clocks are:

- International Atomic Time (TAI), epoch is 1^{st} of January, 1958.
- Coordinated Universal Time (UTC), epoch is 1^{st} of January, 1970
- Global Positioning System (GPS), 1^{st} of January, 1980

There are two ways for achieving the synchronized acquisition:

- acquisition controlled by synchronization source clock
- free-running acquisition clock

The synchronization source provides the accurate time synchronization (around hundreds of nanoseconds). The output of the synchronization source is a square TTL-level digital signal that generates the pulse-per-second (PPS), namely the accurate information about when the UTC second starts. Transistor–transistor logic (TTL) is a logic family built from bipolar junction transistors.

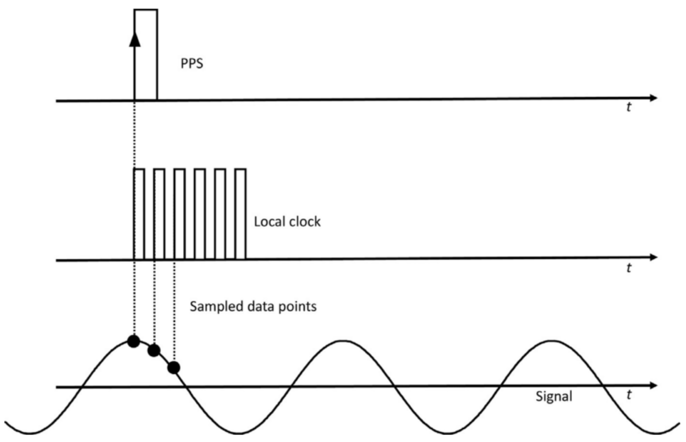


Figure 3.2: Data Acquisition disciplined with 1 PPS [13]

Local clock is the time reference dissemination inside the PMU and that time source is used for the control of the acquisition process. PPS triggers the local clock of the PMU and the measurement process starts at a precise time instant. The local clock has to be locked to the PPS, so that both the alignment and the pace of the acquisition are correlated to the UTC, as it can be seen from the figure 3.2. After the locking is obtained, the samples are acquired on known time instants.

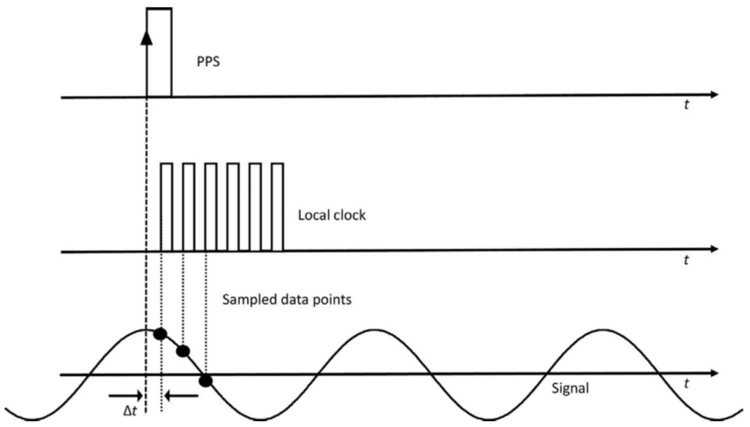


Figure 3.3: Free-running approach for data acquisition[13]

In case when the local clock for the DAQ is not accessible or it is not tuneable from external synchronization sources the only possible solution for the acquisition is less accurate free-running approach (3.3).

In this case local clock is not locked to PPS, causing the offset between the reference time of the measurement and the nearest sample.

3.1.3. Phasor Data Concentrator and Communication

The PDC correlates synchrophasor measurements by the time tag to create a system wide set of measurements. Following that, the set is fed out as a single stream to the higher level PDCs and applications such as EMS and SCADA. Synchrophasor message format for real-time data transmission is specified by the IEEE standard for synchrophasor data transfer. Message frameworks consists of four message types: data, configuration, header, and command.

- Data messages are the phasor quantity estimates obtained from a PMU.
- Configuration message is a machine-readable message describing the data types and calibration factors.
- Header information is human readable descriptive information sent from the PMU/PDC.
- Commands are machine-readable codes for control or configuration.

The frame starts with SYNC, FRAMESIZE, IDCODE, SOC and FRACSEC, and terminates with a CHK as shown in figure 3.4. Since the synchrophasor measurements are tagged with the UTC time corresponding to the time of measurement, the message consisting the information about the time-tag consists a second-of-century (SOC) count and a fraction-of-second (FRACSEC) count. The reporting times shall be evenly spaced through each second with the time of the first frame coincident with the 1PPS. The word SYNC provides synchronization and frame identification, whereas word IDCODE directly identifies the source of a data, header, configuration message and the destination of a command message. All frames ends with the check word (CHK) which is a redundancy check assuring data integrity. The frame starts with SYNC, FRAMESIZE, IDCODE, SOC and FRACSEC, and terminates with a CHK as shown in figure 3.4. A configuration frame is a machine-readable set

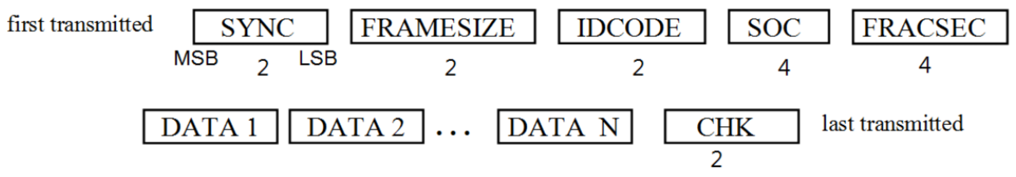


Figure 3.4: Frame transmission order [16]

of the data that contains information and processing parameters for a synchrophasor measurement stream.

4 IEEE Standard C37.118 Compliance

IEEE Standard for Synchrophasor Measurements for Power Systems sets the requirements for the performance of the PMU deployed to monitor the power grid. The standard defines steady-state and dynamic state compliance tests that evaluate accuracy of the PMU measurements under different input signals influenced by grid conditions. The performance of PMU also depends on the presents of higher frequency components and the noise in the input signal. However, proper filtering can reject undesirable signal components within the limits provided by the filter attenuation. The main parameters evaluating compliance with the standard, as well as criteria and limits proposed by standard are represented in the following sections.

4.1. Synchrophasor Measurement Evaluation

The accuracy of the estimated phase and amplitude of the phasor is evaluated by the quantity called Total Vector Error (TVE). If X_{ref} is the reference signal at the input of the PMU, and X_{est} is its estimated value, total vector error can be calculated by following equation:

$$TVE = \sqrt{\frac{(Re[X_{est}] - Re[X_{ref}])^2 + (Im[X_{est}] - Im[X_{ref}])^2}{(Re[X_{ref}])^2 + (Im[X_{ref}])^2}} \tag{4.1}$$

where $Re[X_{ref}]$ and $Im[X_{ref}]$ are real and imaginary part of the reference phasor, whereas $Re[X_{est}]$ and $Im[X_{est}]$ represent real and imaginary part of the estimated phasor.

4.2. Synchrophasor Frequency and ROCOF Evaluation

Similarly to Total Vector Error, frequency and ROCOF errors can be calculated by subtracting the theoretical and estimated values and taking the absolute value of the difference.

Frequency measurement error can be calculated as:

$$FE = |f_{ref} - f_{est}| \tag{4.2}$$

ROCOF measurement error can be calculated as:

$$RFE = \frac{\partial f_{ref}}{\partial t} - \frac{\partial f_{est}}{\partial t} \tag{4.3}$$

4.3. Measurement Reporting Rate

The estimated phasor, frequency and ROCOF should be reported together at constant rate defined by standard. The specified rates Fs for 50 Hz systems are 10, 25 or 50 frames/s, where the frame stands for a set of synchrophasor, frequency and ROCOF that corresponds to the same time stamp. The actual rate to be used should be user adjustable.

4.4. Measurement Reporting Latency Requirements

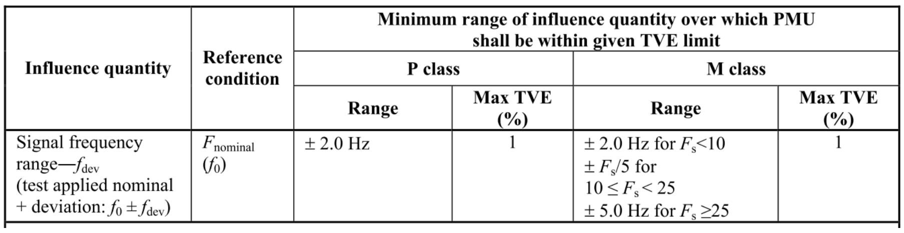
Latency in the measurement reporting is the time between the moment when an event occurs in the power system and the moment when it is reported through the data. This latency includes many factors such as the length of the observing window for DFT estimation, complexity of the phasor estimation technique, PMU processing power, applied filtering technique and when the event occurs within the reporting interval. In the standard for Synchrophasor Measurement, PMU reporting latency is defined as the maximum time interval between the data report time (data time stamp) and the time when the data becomes visible at the PMU output.

4.5. Performance classes

Compliance with the standard requirements is evaluated depending on the class of the performances. This standard defines two classes of the performances: M class and P class. M class in intended for applications that do not require the fastest reporting speed. However, since the letter M stands for monitoring and analytic measurements, this class requires greater precision in phasor estimation.
P class in intended for protection applications requiring fast response and minimal filtering.
A PMU has to meet all the requirements specified for a particular class, in order to be considered as compliant with the standard for that class. If both classes are provided, these should be user selective.

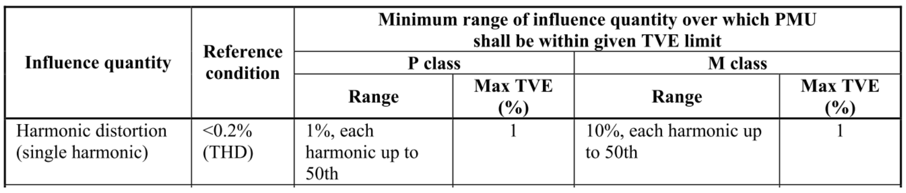
4.6. Steady-State Compliance

In order to test the performance of the PMU during the steady-state condition in electrical grid, amplitude A, frequency f and phase ϕ_0 of the test signal are fixed for the period of a measurement. This includes nominal and off-nominal frequencies(quasi steady-state). The compliance requirements depends on the performance class the application needs and on the purity of the test signal. Requirements for M class and P class with and without harmonic distortion present in the signal are shown in the figures 4.1 and 4.2.



Influence quantity	Reference condition	Minimum range of influence quantity over which PMU shall be within given TVE limit			
		P class		M class	
		Range	Max TVE (%)	Range	Max TVE (%)
Signal frequency range—f_{dev} (test applied nominal + deviation: $f_0 \pm f_{dev}$)	$F_{nominal}$ (f_0)	± 2.0 Hz	1	± 2.0 Hz for F_s<10 ± F_s/5 for $10 \leq F_s < 25$ ± 5.0 Hz for $F_s \geq 25$	1

Figure 4.1: Steady-state synchrophasor measurement requirements with the harmonic distortion [6]

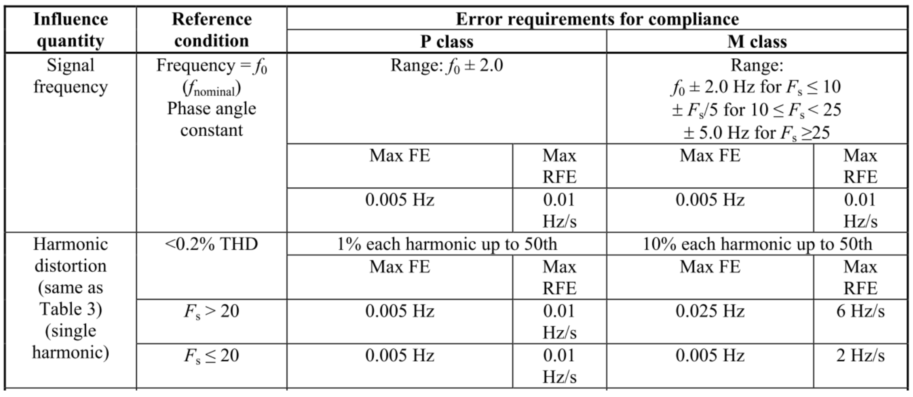


Influence quantity	Reference condition	Minimum range of influence quantity over which PMU shall be within given TVE limit			
		P class		M class	
		Range	Max TVE (%)	Range	Max TVE (%)
Harmonic distortion (single harmonic)	<0.2% (THD)	1%, each harmonic up to 50th	1	10%, each harmonic up to 50th	1

Figure 4.2: Steady-state synchrophasor measurement requirements [6]

It can be seen that maximum allowed TVE is 1 % for both pure and distorted signal.

On the other hand, when it comes to allowed frequency and ROCOF errors the limits are set as per following table:



Influence quantity	Reference condition	Error requirements for compliance			
		P class		M class	
Signal frequency	Frequency = f_0 (f_{nominal}) Phase angle constant	Range: f_0 ± 2.0		Range: f_0 ± 2.0 Hz for $F_s \leq 10$ ± $F_s/5$ for $10 \leq F_s < 25$ ± 5.0 Hz for $F_s \geq 25$	
		Max FE	Max RFE	Max FE	Max RFE
		0.005 Hz	0.01 Hz/s	0.005 Hz	0.01 Hz/s
Harmonic distortion (same as Table 3) (single harmonic)	<0.2% THD	1% each harmonic up to 50th		10% each harmonic up to 50th	
		Max FE	Max RFE	Max FE	Max RFE
	$F_s > 20$	0.005 Hz	0.01 Hz/s	0.025 Hz	6 Hz/s
	$F_s \leq 20$	0.005 Hz	0.01 Hz/s	0.005 Hz	2 Hz/s

Figure 4.3: Steady-state frequency and ROCOF requirements [6]

4.7. Dynamic Compliance

Due to the increased intermittent generation and difficulties to maintain the balance with the consumption, power system is rarely in steady-state. Consequently, in order to test PMU performance under the dynamic conditions, the parameters of the test signal will be changed and varied accordingly. In the next three sections different test conditions will be explained.

4.7.1. Frequency Ramp

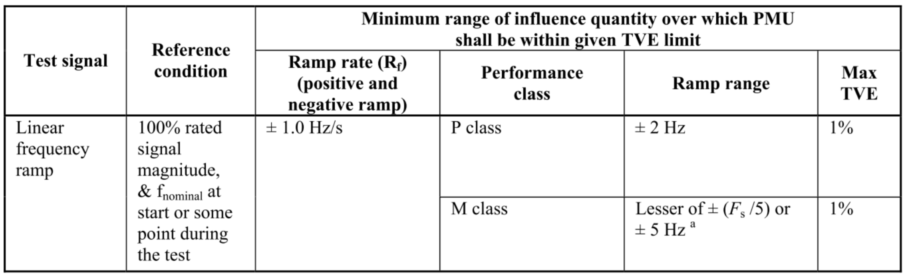
Measurement performance during the change in frequency is tested with linear ramp of the system frequency applied as balanced three-phase input signals. Following equations represents the mathematical test formulation:

$$X_a = X_m cos[\omega t + \pi R_f t^2] \tag{4.4}$$

$$X_b = X_m cos[\omega t - 2\pi/3 + \pi R_f t^2] \tag{4.5}$$

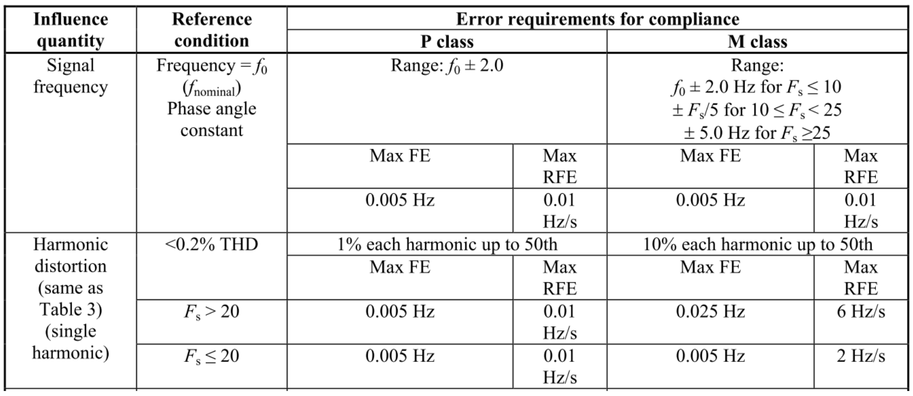
$$X_c = X_m cos[\omega t + 2\pi/3 + \pi R_f t^2] \tag{4.6}$$

where R_f defines the change in frequency $\frac{df}{dt}$ and its value is specified to be 1Hz/s with the ramp range of +/- 2Hz. Estimated frequency and ROCOF error and TVE limits during the testing under these conditions are shown in following tables:



Test signal	Reference condition	Minimum range of influence quantity over which PMU shall be within given TVE limit			
		Ramp rate (R_f) (positive and negative ramp)	Performance class	Ramp range	Max TVE
Linear frequency ramp	100% rated signal magnitude, & f_{nominal} at start or some point during the test	± 1.0 Hz/s	P class	± 2 Hz	1%
			M class	Lesser of ± (F_s /5) or ± 5 Hz [a]	1%

Figure 4.4: Synchrophasor performance requirements under frequency ramp tests [6]



Influence quantity	Reference condition	Error requirements for compliance			
		P class		M class	
Signal frequency	Frequency = f_0 (f_{nominal}) Phase angle constant	Range: $f_0 \pm 2.0$		Range: $f_0 \pm 2.0$ Hz for $F_s \leq 10$ $\pm F_s/5$ for $10 \leq F_s < 25$ ± 5.0 Hz for $F_s \geq 25$	
		Max FE	Max RFE	Max FE	Max RFE
		0.005 Hz	0.01 Hz/s	0.005 Hz	0.01 Hz/s
Harmonic distortion (same as Table 3) (single harmonic)	<0.2% THD	1% each harmonic up to 50th		10% each harmonic up to 50th	
		Max FE	Max RFE	Max FE	Max RFE
	$F_s > 20$	0.005 Hz	0.01 Hz/s	0.025 Hz	6 Hz/s
	$F_s \leq 20$	0.005 Hz	0.01 Hz/s	0.005 Hz	2 Hz/s

Figure 4.5: Frequency and ROCOF performance requirements under frequency ramp tests[6]

4.7.2. Measurement Bandwidth

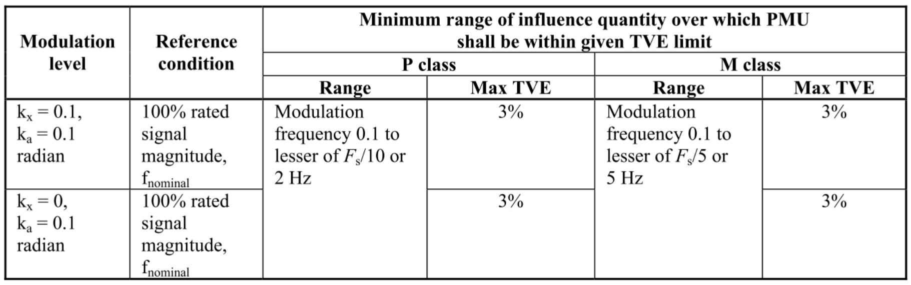
By sweeping the input with sinusoidal amplitude and phase modulation dynamic compliance for a measurement bandwidth can be determined. The mathematical representation of the input test signal for three phases can be represented as following:

$$X_a = X_m[1 + k_x \cos(\omega t)] \times cos[\omega_0 t + k_a \cos(\omega t - \pi)] \tag{4.7}$$

$$X_b = X_m[1 + k_x \cos(\omega t)] \times cos[\omega_0 t - 2\pi/3 + k_a \cos(\omega t - \pi)] \tag{4.8}$$

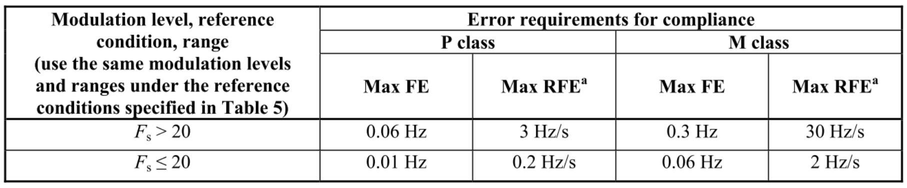
$$X_c = X_m[1 + k_x \cos(\omega t)] \times cos[\omega_0 t + 2\pi/3 + k_a \cos(\omega t - \pi)] \tag{4.9}$$

where ω_0 represents nominal angular frequency, ω modulation frequency, whereas k_x and k_a are amplitude and phase modulation factors respectively. Estimated frequency and ROCOF error and TVE limits during the testing the PMU performance under the condition of amplitude and phase modulation are shown in following tables:



Modulation level	Reference condition	Minimum range of influence quantity over which PMU shall be within given TVE limit			
		P class		M class	
		Range	Max TVE	Range	Max TVE
$k_x = 0.1$, $k_a = 0.1$ radian	100% rated signal magnitude, f_{nominal}	Modulation frequency 0.1 to lesser of $F_s/10$ or 2 Hz	3%	Modulation frequency 0.1 to lesser of $F_s/5$ or 5 Hz	3%
$k_x = 0$, $k_a = 0.1$ radian	100% rated signal magnitude, f_{nominal}		3%		3%

Figure 4.6: Synchrophasor performance requirements under amplitude and/or phase modulation tests [6]



Modulation level, reference condition, range (use the same modulation levels and ranges under the reference conditions specified in Table 5)	Error requirements for compliance			
	P class		M class	
	Max FE	Max RFE[a]	Max FE	Max RFE[a]
$F_s > 20$	0.06 Hz	3 Hz/s	0.3 Hz	30 Hz/s
$F_s \leq 20$	0.01 Hz	0.2 Hz/s	0.06 Hz	2 Hz/s

Figure 4.7: Frequency and ROCOF performance requirements under amplitude and/or phase modulation tests [6]

4.7.3. Step changes in Phase and Magnitude

Performance under step changes in phase and magnitude can be checked by applying balanced three-phase step changes to balanced three-phase input signals. It can be modeled with the equations stated below:

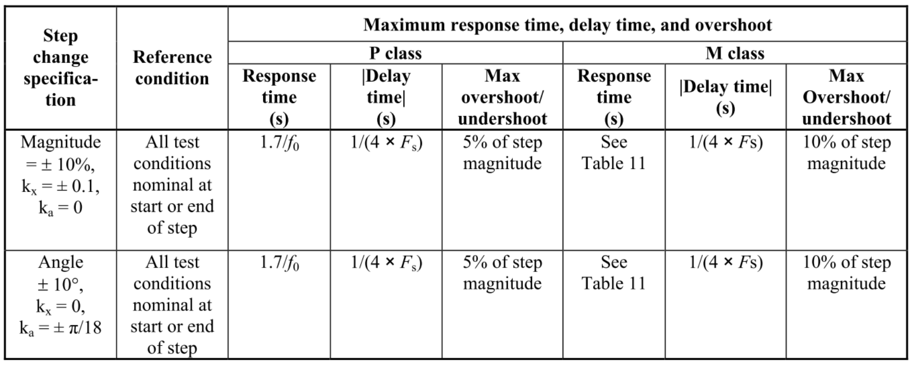
$$X_a = X_m[1 + k_x f_1(t)] \times cos[\omega_0 t + k_a f_1(t)] \tag{4.10}$$

$$X_b = X_m[1 + k_x f_1(t)] \times cos[\omega_0 t + k_a f_1(t)] \tag{4.11}$$

$$X_c = X_m[1 + k_x f_1(t)] \times cos[\omega_0 t + k_a f_1(t)] \tag{4.12}$$

where X_m is the amplitude of the input signal, ω_0 is nominal frequency, k_x and k_a are amplitude and phase step size respectively, $f_1(t)$ is a unit step function. This test represents a transition from one steady state to another and can be used to determine response time, delay time and overshoot in the measurements. [1]

The values of all test input signal parameters and performance requirements for synchrophasor, frequency and ROCOF estimation are shown in following tables:



Step change specification	Reference condition	Maximum response time, delay time, and overshoot					
		P class			M class		
		Response time (s)	|Delay time| (s)	Max overshoot/ undershoot	Response time (s)	|Delay time| (s)	Max Overshoot/ undershoot
Magnitude = ± 10%, $k_x = \pm 0.1$, $k_a = 0$	All test conditions nominal at start or end of step	$1.7/f_0$	$1/(4 \times F_s)$	5% of step magnitude	See Table 11	$1/(4 \times Fs)$	10% of step magnitude
Angle ± 10°, $k_x = 0$, $k_a = \pm \pi/18$	All test conditions nominal at start or end of step	$1.7/f_0$	$1/(4 \times F_s)$	5% of step magnitude	See Table 11	$1/(4 \times Fs)$	10% of step magnitude

Figure 4.8: Synchrophasor performance requirements under input step change [6]

5

Synchrophasor Estimation Based on Smart Discrete Fourier Transform (SDFT)

In this chapter, synchrophasor estimation technique called Smart Discrete Fourier Transform is mathematically explained and verified with the set of test conditions performed in programming environment MATLAB.

5.1. SDFT Technique

SDFT method for phasor estimation is based on mathematical approach, that efficiently solves limitations of basic DFT method. Namely, the method relies on frequency estimation obtained from three consecutive DFT fundamental components. By using estimated frequency, one SDFT phasor is obtained, having significantly higher accuracy then previously estimated DFT phasors. The SDFT algorithm requires additional computational steps compared to the standard DFT, but the benefits of reduced filtering requirements outweigh additional processing expenses.
The proposed algorithm [12] starts with a deployment of a basic DFT method on a discrete sinusoidal signal $x(k)$ in order to extract its fundamental frequency component $\hat{x_r}$. Following equations illustrate that:

$$x(k) = X_m cos(\frac{2\pi f k}{N f_0} + \phi) \tag{5.1}$$

being: $x(k)$ a pure sinusoidal signal sampled at discrete instants, X_m the input signal amplitude, f the signal frequency, ϕ the initial phase angle, f_0 the nominal signal frequency and N the sampling rate in samples/cycle.
This representation can be further modified to:

$$x(k) = \frac{1}{2}(X_m e^{\frac{2\pi f k}{N f_0}+\phi} + X_m e^{-(\frac{2\pi f k}{N f_0}+\phi)}) = \frac{1}{2}(\bar{x} e^{\frac{2\pi f k}{N f_0}} + \bar{x}^* e^{-(\frac{2\pi f k}{N f_0})}) \tag{5.2}$$

where $\bar{x} = X_m e^{j\phi}$ and $\bar{x}^*$ are complex conjugate pair.
The main frequency component in DFT spectrum of $x(k)$, evaluated at the r^{th} sample is given as:

$$\hat{x_r} = \frac{2}{N}\sum_{k=0}^{N-1} x(k+r) e^{-j\frac{2\pi k}{N}} \tag{5.3}$$

If we consider the system frequency deviation to be Δf:

$$f = f_0 + \Delta f \tag{5.4}$$

After substituting 5.2 into 5.3 and performing some algebraic manipulations, following relations can be derived:

$$\hat{x_r} = A_r + B_r \tag{5.5}$$

being:

$$A_r = \frac{\hat{x}}{N} e^{j\frac{2\pi(f_0+\Delta f)r}{Nf_0}} e^{j\frac{\pi(N-1)\Delta f}{Nf_0}} \frac{\sin(\frac{\pi\Delta f}{f_0})}{\sin\frac{\pi\Delta f}{Nf_0}} \tag{5.6}$$

$$B_r = \frac{\hat{x}^*}{N} e^{j\frac{2\pi(f_0+\Delta f)r}{Nf_0}} e^{j\frac{\pi(N-1)(f+\Delta f)}{Nf_0}} \frac{\sin(\frac{\pi(f_0+\Delta f)}{f_0})}{\sin\frac{\pi(f_0+\Delta f)}{Nf_0}} \tag{5.7}$$

If we replace the exponential kernel in 5.6 and 5.7 with parameter a as:

$$a = e^{j\frac{2\pi(f_0+\Delta f)r}{Nf_0}} \tag{5.8}$$

the following relations can be obtained:

$$w = a + a^{-1} = 2\cos\frac{2\pi(f_0+\Delta_f)}{Nf_0} \tag{5.9}$$

So, the frequency deviation Δf can be derived as:

$$\Delta f = \frac{Nf_0}{2\pi}\cos^{-1}(\frac{w}{2}) - f_0 \tag{5.10}$$

The phasor can be estimated by rearranging 5.6 as:

$$A_r = \frac{X_m}{N} \frac{\sin\frac{\pi\Delta f}{f_0}}{\sin\frac{\pi\Delta f}{Nf_0}} e^{(j\frac{2\pi(f_0+\Delta f)r}{Nf_0} + j\frac{\pi(N-1)\Delta f}{Nf_0} + \phi)} \tag{5.11}$$

Now, amplitude X_m and phase ϕ can be extracted:

$$X_m = |Ar| \frac{N\sin\frac{\pi\Delta f}{Nf_0}}{\sin\frac{\pi\Delta f}{f_0}} \tag{5.12}$$

$$\phi = angle(A_r) - \frac{\pi(N-1)\Delta f}{Nf_0} \tag{5.13}$$

However, since the phasor rotates when the data window moves by one sample, the phasor angle estimate should be corrected to obtain a stationary phasor:

$$\phi = angle(A_r) - \frac{\pi(N-1)\Delta f}{Nf_0} - \frac{2\pi}{N}m \tag{5.14}$$

being m a counter varies from 0 to (N-1).
In order to find the value of w and hence the unknown Δf, three consecutive DFT fundamental components $\hat{x_r}$, $\hat{x_{r-1}}$, $\hat{x_{r-2}}$ are considered:

$$\hat{x_r} = A_r + B_r \tag{5.15}$$

$$\hat{x_{r+1}} = A_{r+1} + B_{r+1} = aA_r + a^{-1}B_r \tag{5.16}$$

$$\hat{x_{r+2}} = A_{r+2} + B_{r+2} = a^2A_r + a^{-2}B_r \tag{5.17}$$

After the algebraic manipulations with the equations 5.15, 5.16, 5.17 and 5.9 and expression for w can be derived as:

$$w = \frac{\hat{x_r} + \hat{x_{r+2}}}{\hat{x_{r+1}}} \tag{5.18}$$

From 5.15, 5.16 and 5.17, parameter A_r can be calculated as:

$$A_r = \frac{a\hat{x_{r+1}} - \hat{x_r}}{a^2 - 1} \tag{5.19}$$

Finally, after estimating the values of A_r and Δf, SDFT phasor can be obtained.

5.2. Simulations and IEEE Std.C37.118 Compliance Tests

The performance of proposed SDFT algorithm is verified in MATLAB environment. In the later stage of the thesis, the same algorithm is written in programming language Python, adapted for real-time implementation with respect to limitations of the embedded platform. Following pseudo-code, shows the flow of the algorithm implemented in MATLAB:

```
procedure: SDFT-based synchrophasor estimation algorithm
    1. run sliding DFT algorithm for a complete duration of the input signal
    2. by taking three consecutive DFT fundamental components estimated in the previous step, run SDFT
       algoritm for complete duration of the input signal
    3. calculation of TVE and errors in estimated frequency and ROCOF
end procedure
```

The simulations are performed by using a single-phase sine wave input. All conclusions obtained for one phase can be referred to other two phases in the balanced system.

5.2.1. Simulations under the steady-state conditions

The (quasi)steady-state conditions are modeled with the input signal with fixed frequency of 48Hz, 50Hz and 52 Hz separately. Mathematical representation of the signal is:

$$x(t) = X_m cos(2\pi f + \phi) \tag{5.20}$$

Amplitude of the signal is set to be 1V, phase equal to 0 rad and frequency will take three different values specified above. The plot of the input signal is printed as figure 5.1
The sampling rate is chosen to be 4kHz. That means that one cycle of the input signal is represented with 80 discrete values. Finally, in duration of 1 s, there are 4000 samples of the input signal. After specifying the

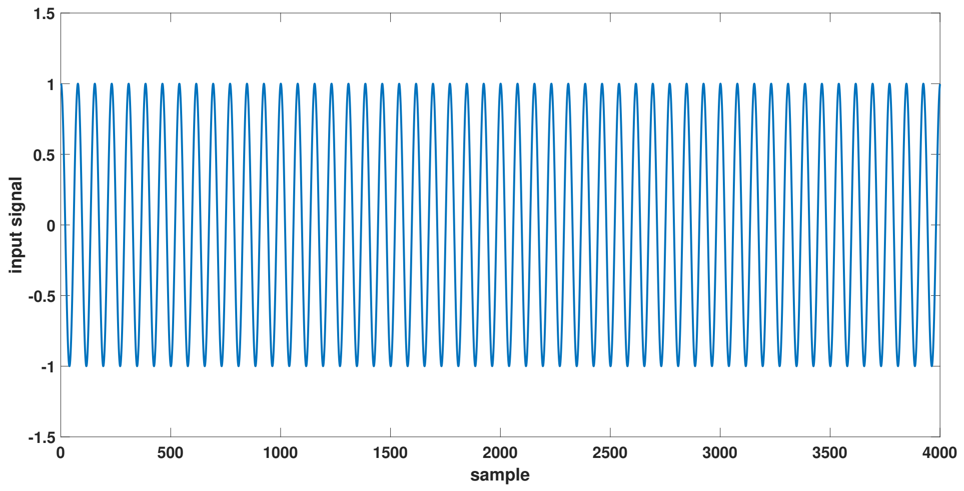


Figure 5.1: Input test signal at frequency of 50Hz

signal parameters, DFT and SDFT part of the code are performed. The results of the simulation are shown in figure 5.2. There are four plots important for evaluation: total vector error, estimated frequency, phase and amplitude calculation for each sliding window. On each plot estimated value is compared with a respective reference or limit. As it can be seen from the figure, the accuracy of the estimation is high, indicated with a low TVE error, approximated to 0%.

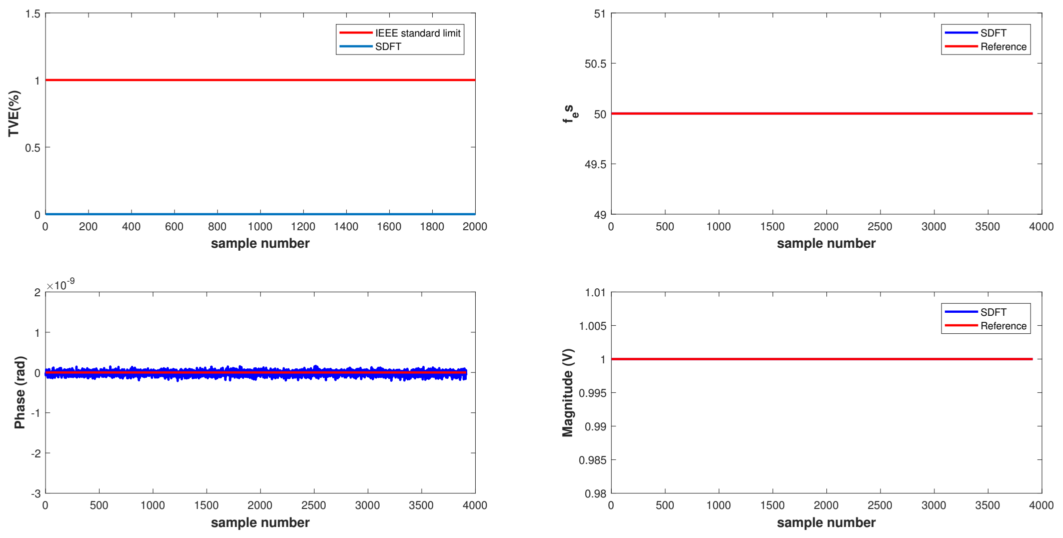


Figure 5.2: The algorithm performance during the steady-state conditions at 50Hz

Similar performance is found under the quasi- steady state conditions when during the 1 s, the frequency has off-nominal value. The results of the simulations are shown on figure 5.3 and figure 5.4.

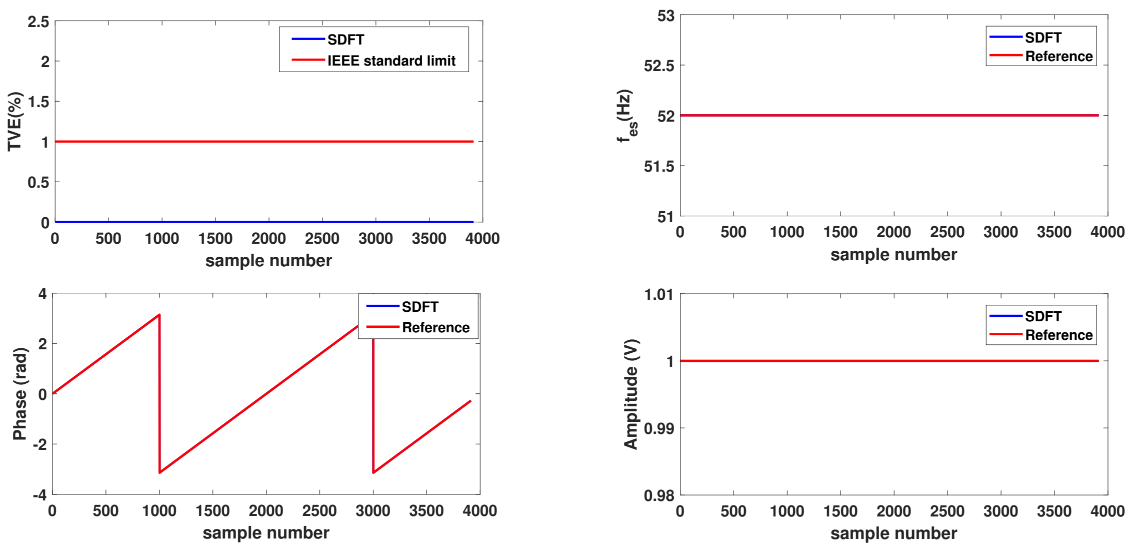


Figure 5.3: The algorithm performance during the steady-state conditions at 52Hz

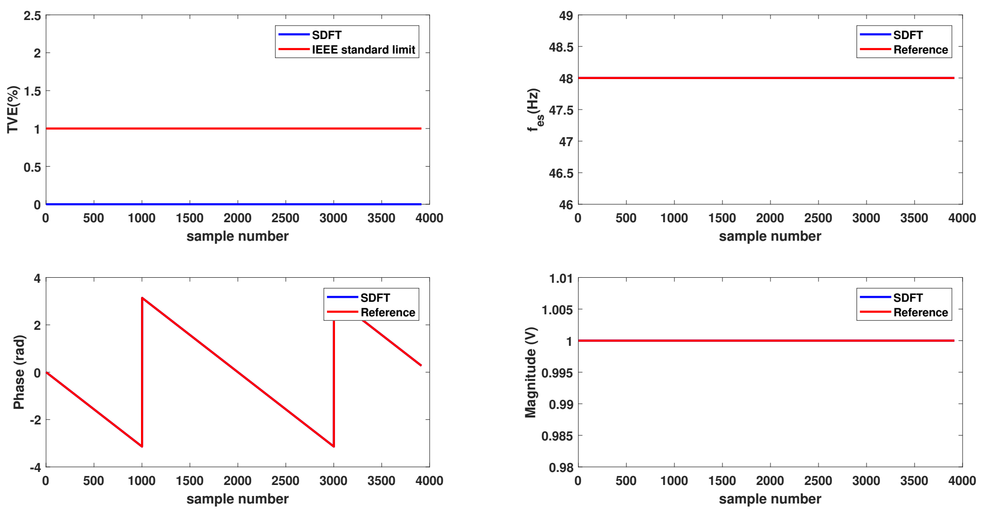


Figure 5.4: The algorithm performance during the steady-state conditions at 48Hz

5.2.2. Simulations under the dynamic state - frequency ramp

An imbalance between available generation and load requirements affects the change in frequency to off-nominal values. The frequency change can be modeled by using a linear function having positive or negative ramp. Since the standard requires compliance under the ramp of +/- 1Hz/s, following input signal is used for estimation algorithm.

$$x(t) = X_m cos(2\pi f + \phi + \pi R_f t^2) \tag{5.21}$$

where R_f represents the ramp in frequency from its nominal value f=50 Hz The results after applying the positive ramp will be reported. By setting the duration of the input signal to 2 s, the frequency is changed from 50 to 52 Hz with the step of 0.0025. The results of the simulation are shown in figure 5.5.

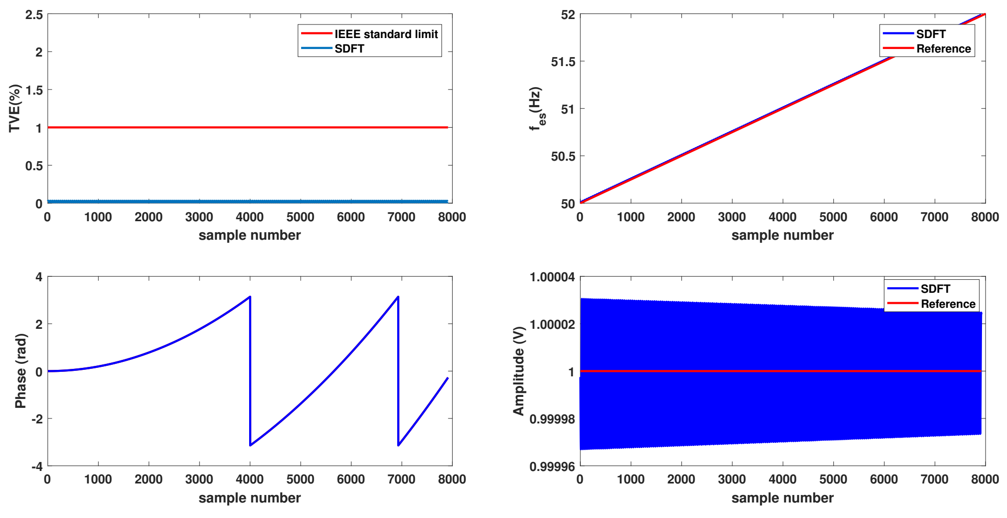


Figure 5.5: The algorithm performance during the frequency ramp change from 50Hz to 52Hz

Even though the frequency vary significantly, the TVE is still within the limits proposed by standard. Namely, the maximum TVE is 0.321% and the minimum TVE is 0.0115%. However, maximum and minimum frequency estimation error is 0.0117 Hz and 0.0085 respectively. Recalling from chapter 4, the maximum allowed frequency error is only 0.005Hz. So the results are not compliant with the standard with respect to the frequency estimation. In the later stage of the thesis, this problem is solved by introducing the mean filter.

5.2.3. Simulations under the dynamic-state conditions - measurement bandwidth

The measurement bandwidth test demonstrates oscillations in the electrical power system. This test includes two tests: amplitude and phase angle modulation. In the magnitude modulation test, 10% modulation signal is added to the signal amplitude and modulation frequency is varied over the range of 0.1 to 5 Hz. The phase angle modulation test is analogue, as 10% modulation signal is applied to the phase angle.

The first test is performed by applying amplitude factor k_x from equation 4.7 equal to 0.1, and modulation frequency f_m of 5Hz. The input signal obtained in this way is shown in figure 5.6.

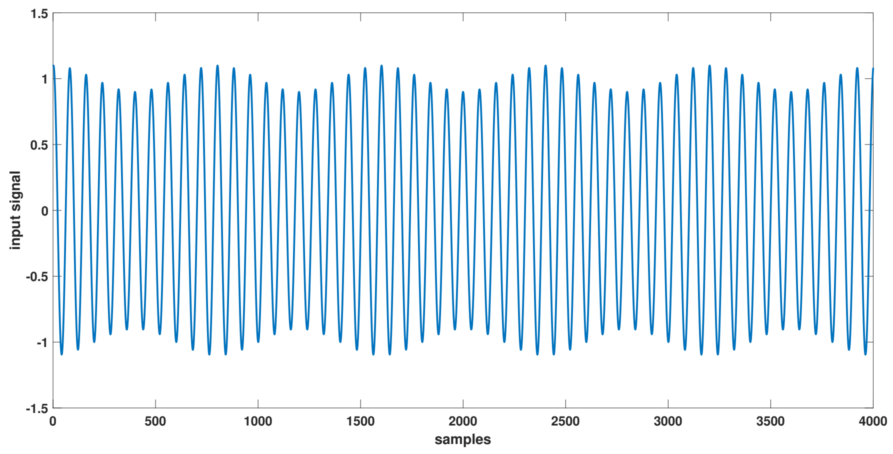


Figure 5.6: Input test signal at frequency of 50Hz with amplitude modulation at 5Hz

Figure 5.7 indicates the estimation accuracy of the algorithm after applying the input signal. Starting from the last plot, it noticeable that the amplitude modulation is well estimated by the algorithm. However, the oscillations in frequency estimation affects the phase calculations. Namely, the frequency estimated by SDFT varies around 50Hz. There is a clearly visible mean value in f_{es} with the amplitude of 50.02 Hz, and oscillations around reaching values over 50.04 Hz. Maximum frequency error occurred is 0.0554 Hz. TVE has the maximal and minimal value equal to 0.663 % and 0.1332 % respectively.

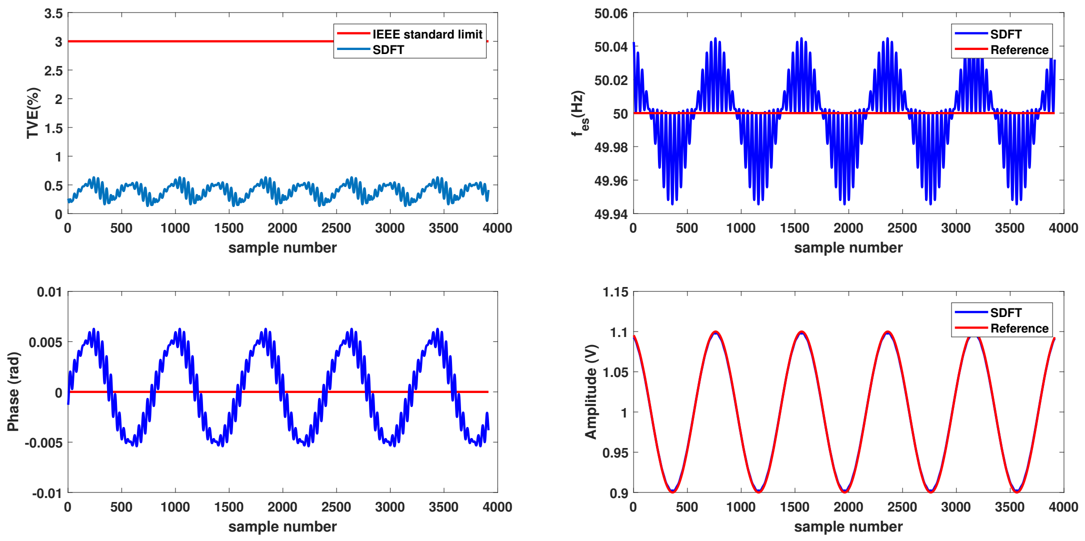


Figure 5.7: The algorithm performance during the amplitude modulation at 5Hz

The simulation is repeated with one difference in initialization, the modulation frequency is reduced to

2 Hz. In this case, better accuracy is obtained. Maximum TVE is 0.2146 %, whereas the maximum frequency error is 0.0088Hz. The plots can be seen in figure 5.8.

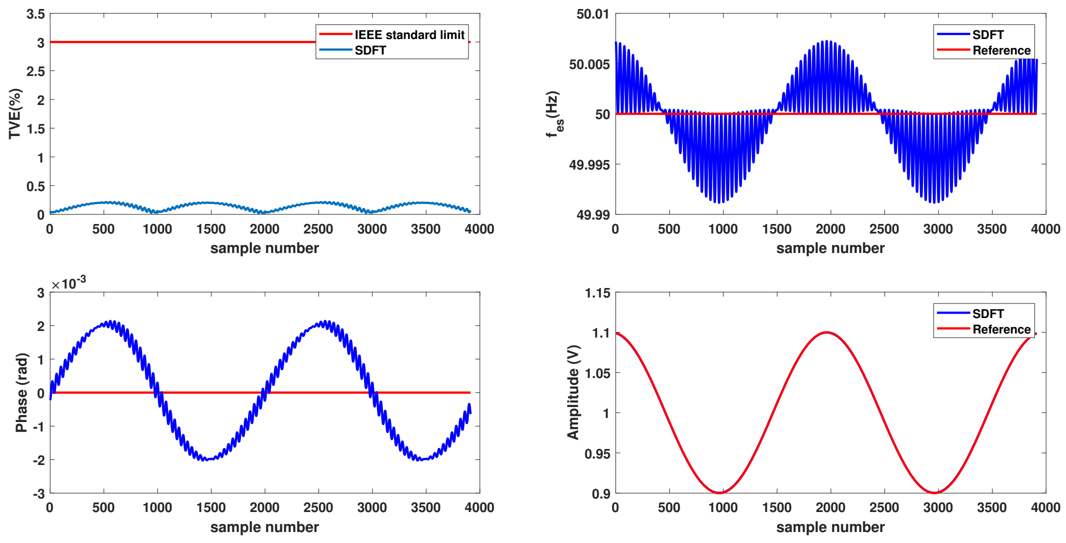


Figure 5.8: The algorithm performance during the amplitude modulation at 2Hz

Following two tests are related to the phase modulation. By using equation 4.7, phase modulation factor is set to 10% and modulation frequency to 5Hz and 2Hz respectively. The figure 5.9 shows the results after applying the sinusoidal change in phase at frequency of 5Hz and 2 Hz respectively. It can be seen that the TVE is below the critical limit of 3%. Maximum value of TVE is 0.4977 % while the minimum value is equal to 0.0016 %. Maximum frequency error is 0.4920 Hz.

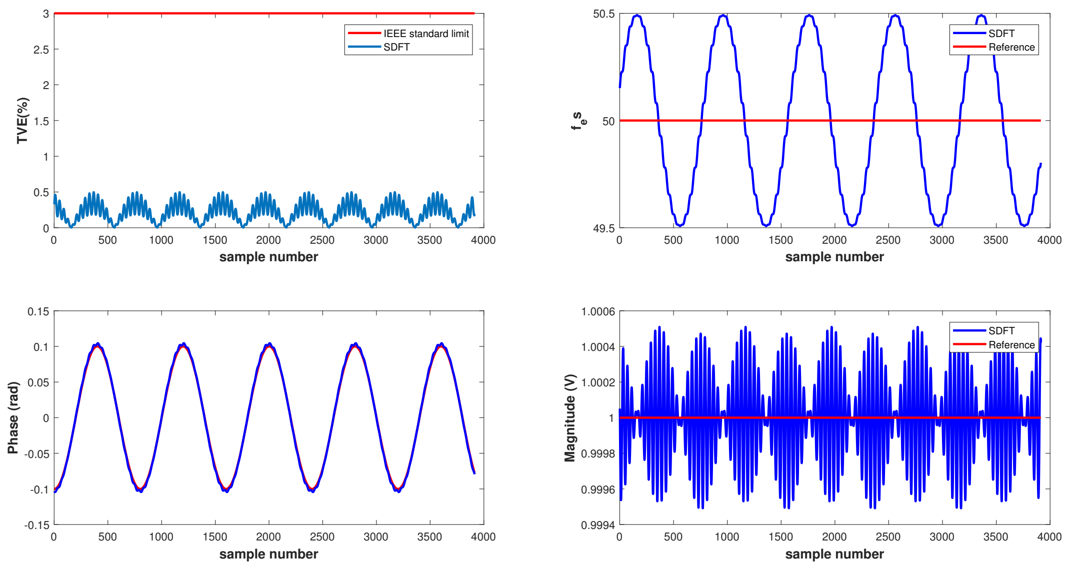


Figure 5.9: The algorithm performance during the phase modulation at 5Hz

When the modulation frequency is reduced to 2 Hz, more accurate estimates are obtained (figure 5.10). Maximum total vector error is 0.008 %, while the maximum frequency error is 0.1995 Hz.

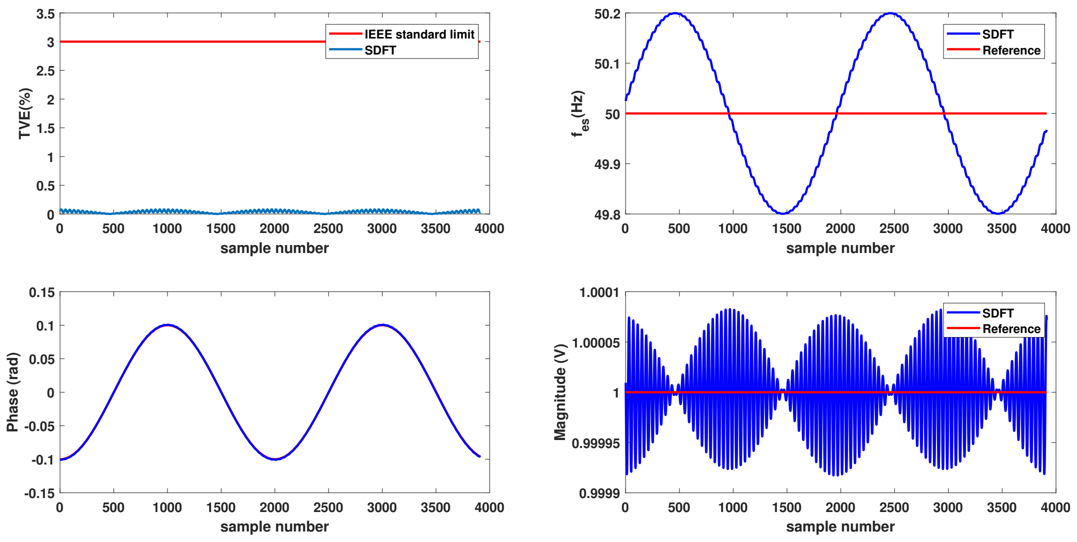


Figure 5.10: The algorithm performance during the phase modulation at 2Hz

The final two cases are referred to as the compliance tests during the amplitude and phase modulation. So, both modulation factors, k_x and k_a are equal to 0.01. The figures 5.11 and 5.12 are showing the results for modulation frequency of 5Hz and 2 Hz respectively. Since both quantities, amplitude and phase vary, the estimation accuracy is slightly lower then in previous cases when only one quantity has been varied, keeping another one constant. If we compare TVE on figure 5.7 and TVE on figure 5.11, the error has been increased from 0.663% to 1.977 %. Likewise, the frequency error has been increased from 0.0554 Hz to 0.4961 Hz, since the phase modulation is also applied. Similar observations are coming from the figure 5.12. Maximal value of TVE is 0.7789 %, while the maximum frequency error is 0.2006 Hz.
In both cases TVE is within the standard limits, whereas the value of frequency error of 0.4951 Hz violates the limit of maximum 0.3Hz.

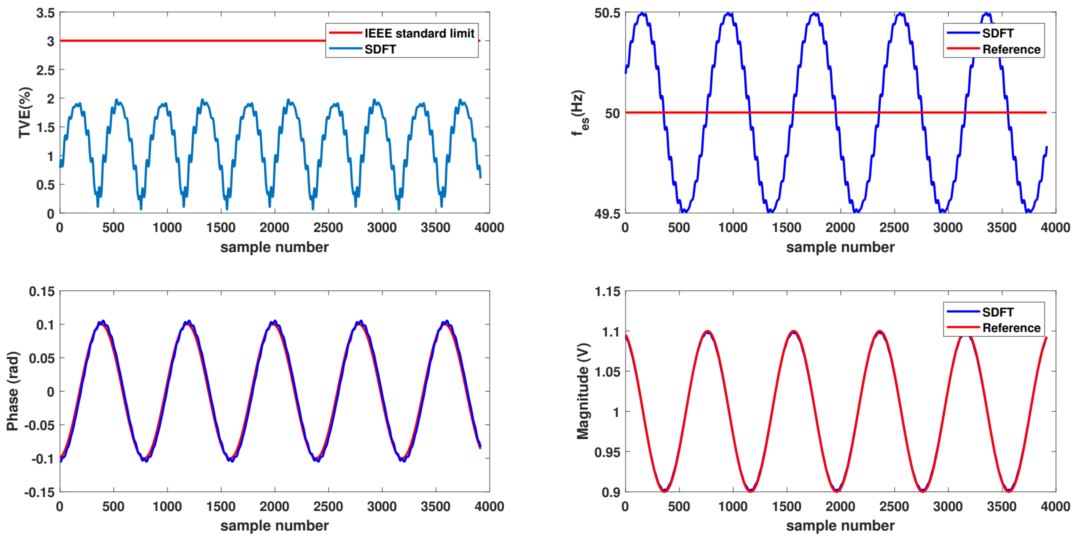


Figure 5.11: The algorithm performance during the amplitude and phase modulation at 5Hz

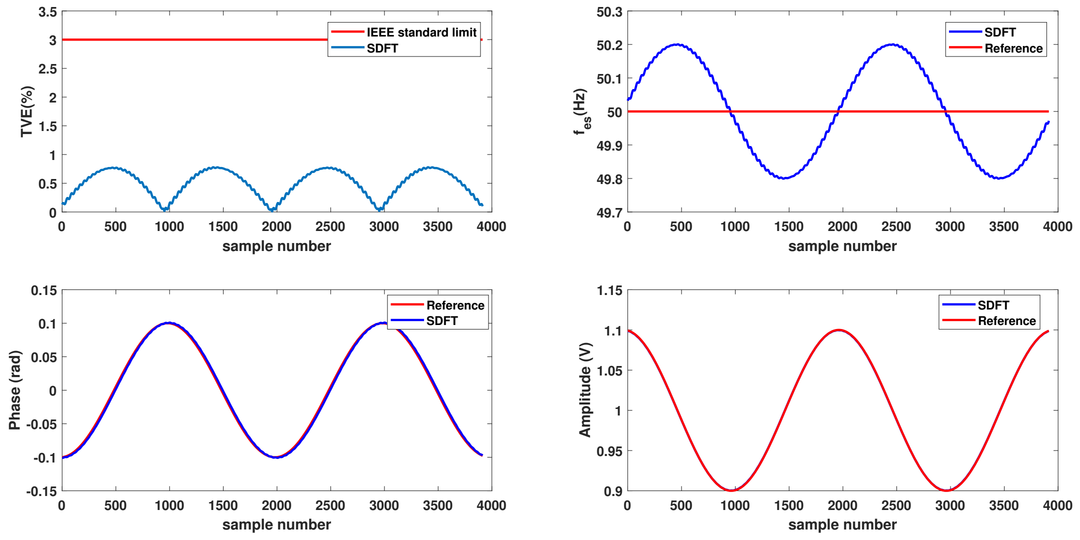


Figure 5.12: The algorithm performance during the amplitude and phase modulation at 2Hz

5.2.4. Simulations under the dynamic-state conditions - step changes in phase and amplitude

The step response tests simulate the power system switching events. There is a magnitude step test, when ±10% step is applied to the signal magnitude, and a phase step test during which ±10 degrees step is applied to the signal phase angle.

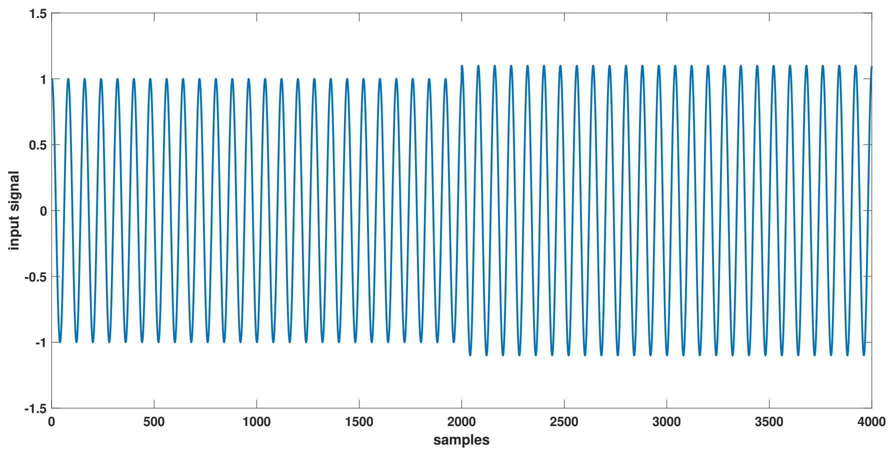


Figure 5.13: The input signal with the amplitude step of 10%

The figure 5.14 shows the results of a magnitude step test. The amplitude value of 1V is increased to 1.1 V at t=0.5 s. Time of half of the second is equivalent to the instant when the 2000th sample is analyzed. At that moment, TVE has been steeply increased to approximately 6 %. A huge overshoot is present in all estimated values around the instant of time when the step occurs. The reason is the length of the observation window of 80 samples in our case. The smaller the length, more accurate estimation during the transients can be achieved.

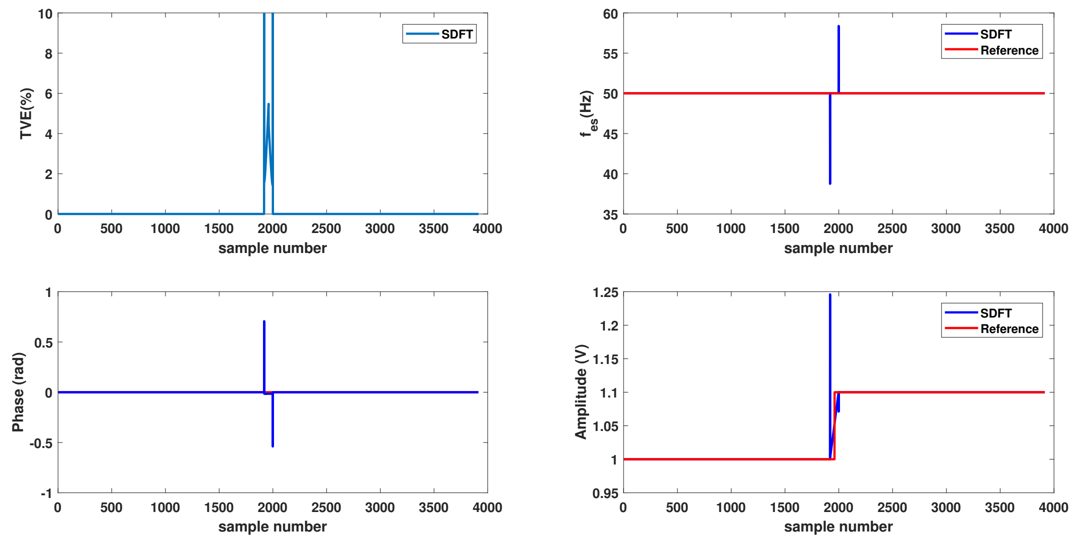


Figure 5.14: The algorithm performance during the amplitude step of 10%

Similar results are obtained during the step in the phase angle. The step of ±10 degrees is equivalent to $\pm\pi/18$ rad. The phase angle is increased from 0 to 0.174 rad at t=0.5s. The maximum TVE overshoot is around 10 %, while estimated frequency error is up to 1.5 Hz.

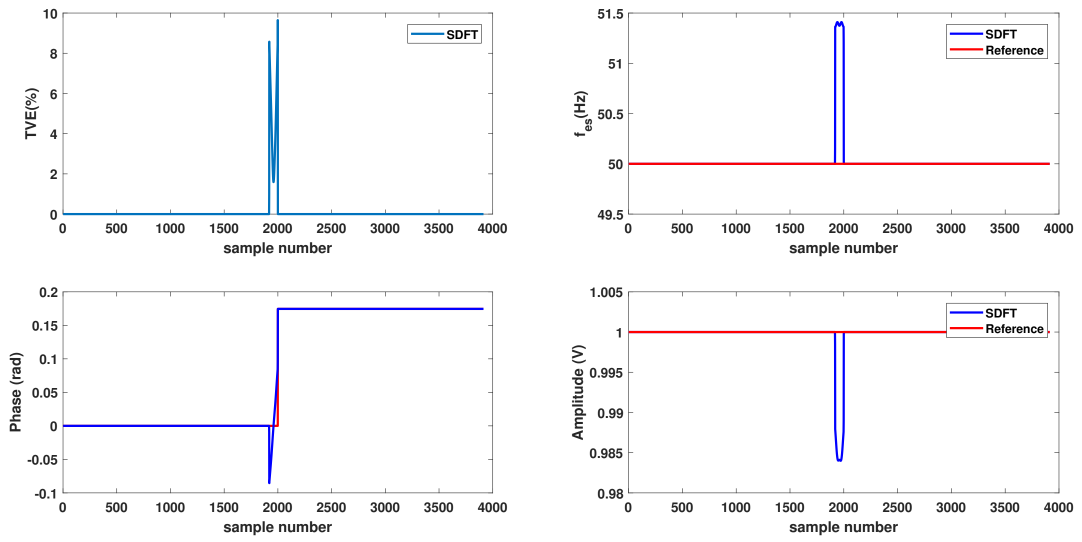


Figure 5.15: The algorithm performance during the phase step of $+\pi/18$

6 LVSensors - low voltage and current measurement sensors

In order to deploy the SDFT algorithm in real-time application, the Master project is merged with the Smart State Technology (SST), the company that develops low voltage measurement devices (LVSensors), as the means for conducting research on Smart Grids. The context of this chapter about the hardware used in the thesis is provided by the engineers in the company, Dr.ir. Omar Mansour and Dr. ir. Dennis Bijwaard.
The idea of the SST is to measure voltages and currents highly synchronized, and at the same time provide high accuracy and resolution. This enables capturing and monitoring grid phenomena and high speed events. The sensors can be easily installed on any location in the distribution grid or in houses after the meter and as such they form a distributed grid monitoring solution.

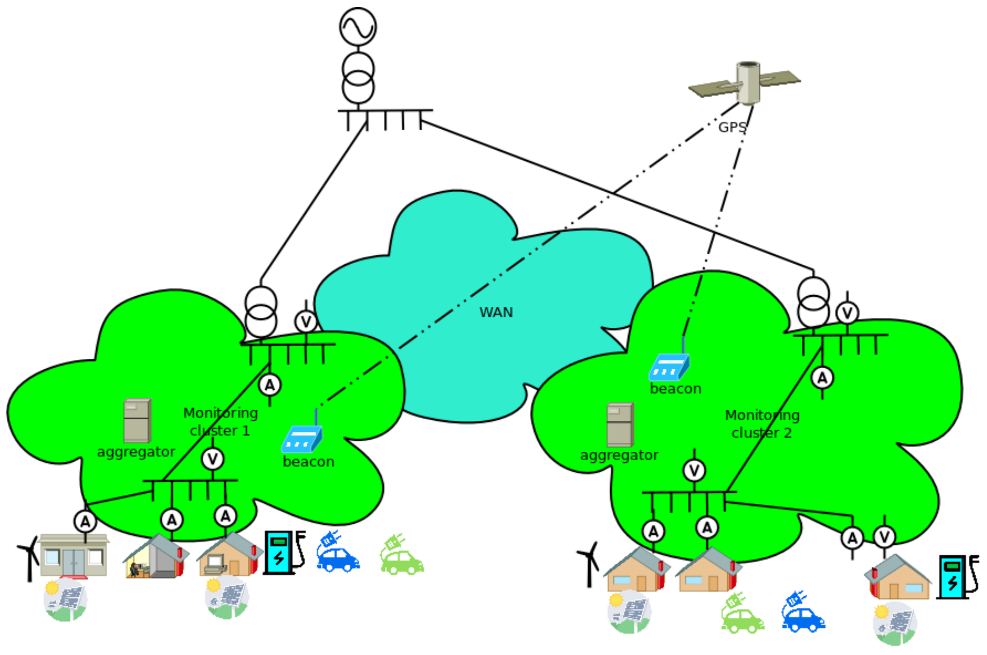


Figure 6.1: LVSensors as a distributed grid monitoring solution

Besides the measurement capability of the sensors, there was the need for users to directly use the real-time measurements in their applications. Using the LVSensors as an open platform gives researchers real access to grid data and real-time measurements. To enable this open platform, SST chose Armbian Linux as an operating system and developed a real-time DSP framework that utilizes the open source messaging system ZeroMQ [17] combined with the structured binary format CBOR[18]. The resulting open platform offers computer-language and operating system independent chaining of DSP algorithms and grid applications.

Sensor measurements are highly synchronized (nanosecond resolution, GPS based time lock) with measurement accuracies of metering grade equipment and high sampling rate capabilities of upto 128 Khz.

This allows for developments in the following SmartGrid domains:

- Remote monitoring (RTU)
- Advanced protection (synchrophasor and state estimation based protection)
- (Distributed) Power quality
- (Dynamic) State estimation
- Congestion detection and Management
- Data analytic
- Control

6.1. System Architecture

In a simple form the system architecture would consist of a time beacon, voltage sensors, current sensors and data aggregation units (which could be a software module installed on one or more sensors or a separate physical module). The aggregated information can be converted to other/standard protocols and sent to a control center.

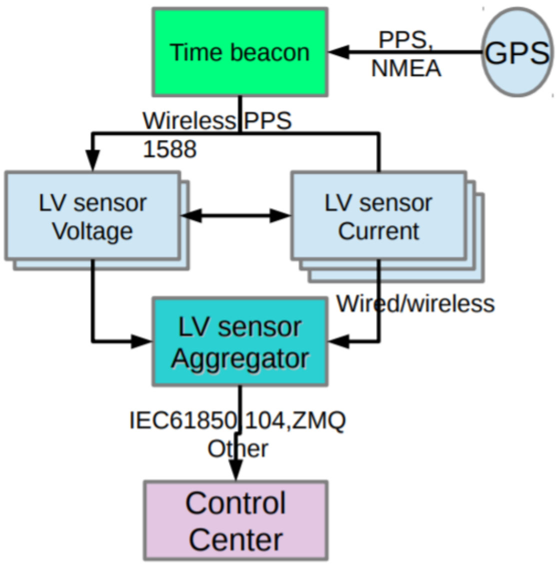


Figure 6.2: LVSensors system architecture

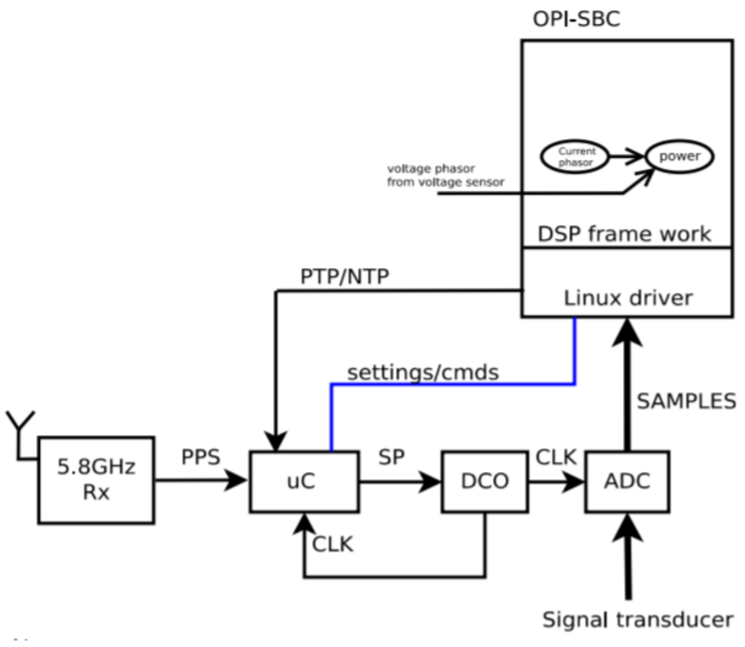


Figure 6.3: Sensor architecture

6.1.1. Time beacon

Time beacon is connected to a modern GPS unit which provides an accurate time pulse (called PPS) signal. The PPS signal is conditioned by the Time-beacon module (figure 6.4) and transmitted wirelessly using a 5.8 GHz analogue video transmitter.

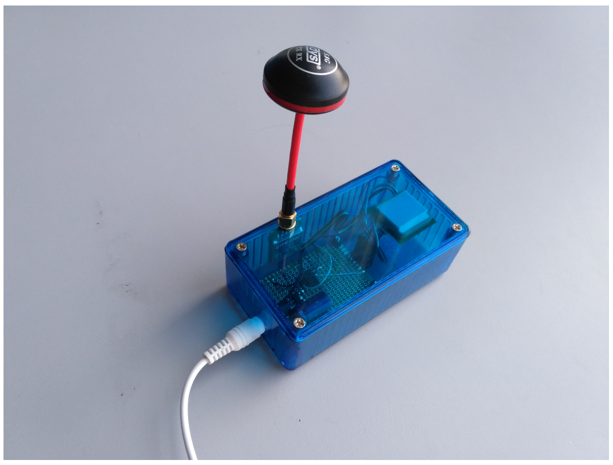


Figure 6.4: Prototype of the time beacon with embedded GPS module and 5.8GHz transmitter

The analogue video transmitter has a constant latency in the order of few hundred nanoseconds and has a range of 1500 meters in open space. In buildings and houses the 5.8 GHz signal range is reduced and less reliable due to obstructions and multi-path effects. In order to circumvent multy-path effects, circular polarized cloverleaf antennas are used [19] in combination with an intelligent estimator at the signal reception side of the sensors.

6.1.2. LV Sensors

The LVSensors come in 2 main flavors, (1) voltage sensors and (2) current sensors. Internally both sensors contain a Raspberry-PI Arm single board computer (SBC) with a shield that contains an advanced multi channel ADC (with simultaneous sampling capability up/to 128KHz), a digitally programmable clock oscillator, a micro-controller and the PPS transceiver circuitry (see fig 6.3). Only the voltage and current transducers of the sensors differ. The voltage sensors have a measurement range of 600V (230V nominal) and make use of signal transformers, while the current sensors make use of split-core CT's and have a measurement range up/to 600A depending on the type of split-core CT used. Rogowski measurement senors are also envisaged and will be added to the LVSensors family in near future.

The micro-controller controls a digital clock oscillator which provides the clock signal for the ADC (which is also the clock of the micro-controller itself) in a feedback loop. It runs an intelligent phase locked loop estimation algorithm to ensure that the acquired samples have high time synchronization to the received GPS-PPS signal. Since the system can not continuously rely on the GPS-PPS signal, the estimation algorithm combines both the PPS signal and a nanosecond accurate time pulse from the SBC, in order to maintain phase and frequency lock in the event of disruption or loss of PPS signal.

Within the SBC, SST has implemented a Linux kernel driver (based on the Industrial-IO framework) for receiving the ADC samples. The default sampling rate of the system is 4KHz (which is sufficient for most grid algorithms and applications). In the case of advanced algorithms that require dedicated sampling, the ADC can be reconfigured for other sampling rates.

The driver will receive the obtained samples from the ADC (via an SPI interface) and makes them available to the user space environment of the Linux system.

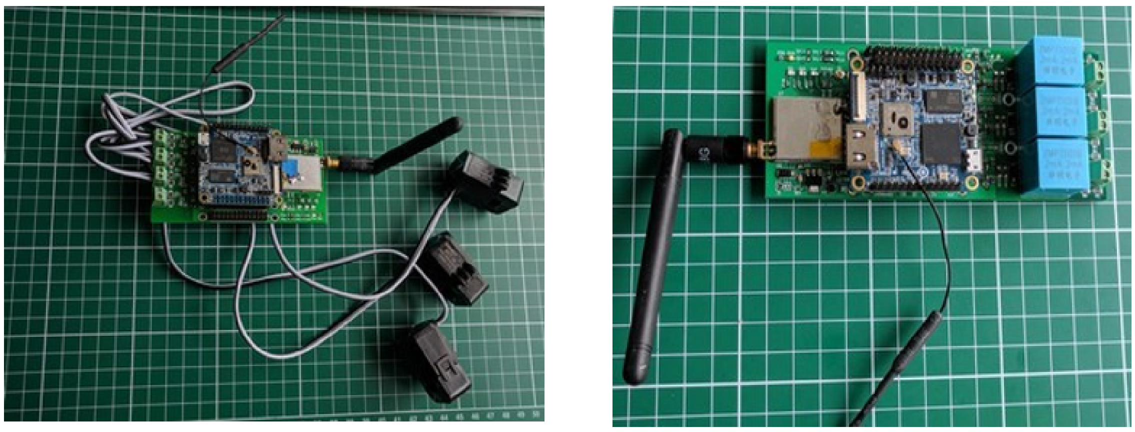


Figure 6.5: LVSensors, current sensor on the left and voltage sensor on the right

6.1.3. DSP framework

A major part of the LVSensors-OpenPlatform concept of is the DSP-framework. The DSP framework ensures that data, measurements and signal events can easily be exchanged between various sensors and aggregation units on distributed locations (on the grid) as well as between various algorithms (in different programming languages) running on a single sensor or different threads running within a single process; all in a uniform programming fashion.

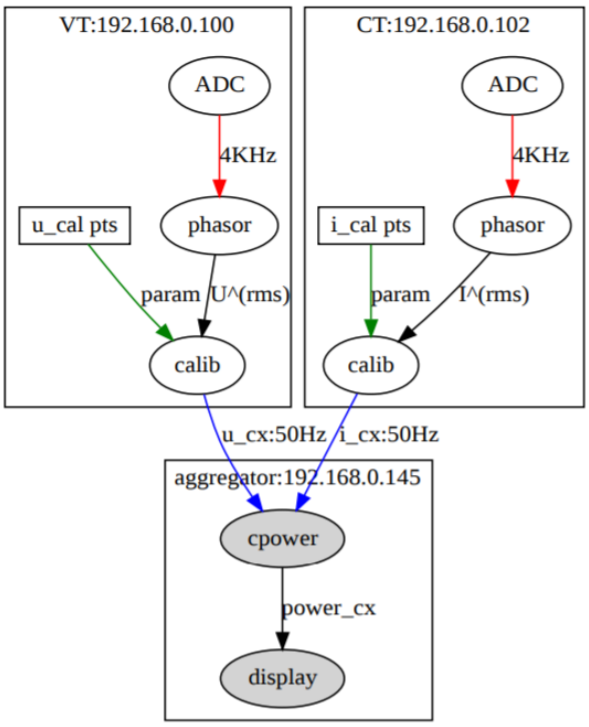


Figure 6.6: Example of a dsp chain for implementation of complex power, running on two separate sensors and an aggregation unit

In the example in fig 6.6, we see how we can use the sensors in a distributed fashion to calculate (for example) the complex power phasors. We use 2 sensors (192.168.0.100 and 192.168.0.102) to calculate the voltage and current phasors. Those phasor signals will be transmitted with a frequency of 50 Hz over IP to an aggregation unit (192.168.0.145) which will essentially calculate and display their complex product.

7 Integration of the SDFT algorithm in embedded platform

In the previous chapter, the performance of the SDFT method has been verified in MATLAB. The following step is the adaptation of the algorithm and and its real-time implementation in embedded sensor. For that purpose, the Python programming language has been chosen.
The code has been split in three main scripts run by sensor:

- pmu_rtd.py
- Window.py
- SDFTWindow.py

and $rtd\_pyPMU.py$ script run by computer. The main script, pmu_rtd.py, defines set of important methods for different purposes such as updating the input with new samples (method $updateInput$, line 62 in Appendix), sending the data to the output ($send\_output$, line 57), TCP and IP configuration for a communication link between the sensor and laptop ($dsp\_realtime$, line 226). Since the sampling frequency of the ADC converter in 4kHz, there are 4000 samples/channel that arrives in period of 1 s. However, the data is reported to sensor in batches of 40 samples, introducing the delay of 10 ms. The data is stored in queue, and by calling the method $updateInput$, the received message can be read and data used for further calculations.
The header of each message consists of following elements:

- tag - name of the sensor as specified in rtd.properties file
- rtdidMain - id of the port input stream is received from
- rtd_items - number of aggregated samples in message
- rtd_len - number of channels within a sample
- seq - sequence number of the first sample in aggregated series
- ts - timestamp of fetching the first sample in the aggregated series
- interval - time between two samples

while the tail of the message carries the samples for each channel (four channels in total). File rtd.properties represents the list of IP adresses and TCP ports needed to enable the communication between the sensor and laptop. Current real-time implementation of the code relies on the sensor for sampling and running the code, while laptop receives the estimates with the script $rtd\_pyPMU.py$ and sends the measurements to PMU Connection Tester. At this point, it is good to clarify the relationship between the scripts $pmu\_rtd.py$ and $rtd\_pyPMU.py$. Namely, they are the scripts with the same methods used for different purposes. The script $rtd\_pyPMU.py$ is using a method $updateInput$ to receive the estimates sent from the sensor. This script is merged with $pyPMU$ implementation of the IEEE C37.118 standard for Data Transfer [20], that sends the data further from the computer to the main PDC (PMU Utility in RSCAD). More details about the communication part will be referred later in the text. In order to perform the DFT and SDFT calculations in real time, it was challenging to find the most appropriate and optimal way to store the available samples. This part is realized by method $add\_samples$ in script Window.py (in Appendix, line 111). There are arrays of 160 length each, storing samples, corresponding sequence and timestamp respectively in batches of 40. The method new_test defines the flow of the DFT and SDFT calculations. The parameter $window\_position$ indicates the end of the data window. Its initial value is 80, since there are 80 samples within one data window covering one cycle of the input signal. In order to perform the sliding DFT, the $window\_position$ is increasing by 1 with each loop. Another important variable is $sample\_count$ indicating the position of the last updated sample in the array. The condition for running the first DFT calculation is that the $sample\_count$ is greater than $window\_position$. That means that there is enough data for one DFT calculation (80 samples). In the early stage of the implementation, phasor estimation, within the each window, was obtained by using non-recursive DFT, meaning that many redundant calculations were being performed. Consequently, the CPU utilization was significantly increased affecting the performance of the sampling module. In order to prevent the lost of the samples, the optimization of the algorithm was performed. Many different ways for minimizing required processing power were considered such as decreased precision in the calculations and decreased number of redundant calculations by using constants. However, the most contributing way was introduction of recursive DFT method. Namely, new parameter named MAX_RECURSIONS was introduced to define the number of recursive DFT calculations performed after one non-recursive. By increasing the value of MAX_RECURSIONS, the computational efficiency of the algorithm was increased. As it is generally known, the recursive approach causes the numerical instabilities, due to the accumulated error from previous estimations. The drawback of that approach is solved by repeating non-recursive DFT after number of recursive estimates. In that way the error is suppressed by resetting the calculations with each non-recursive DFT. SDFT algorithm is realized in the script SDFTWindow.py. In order to decrease CPU load caused by running SDFT calculation, the number of $runSDFT$ executions is reduced to 10, 25 or 50 times per second, depending of the reporting rate. So, once the SDFT estimates(phasors and frequency) are obtained they are directly reported to the output. Together with the estimates, corresponding timestamp is sent. Since the value of the estimated phasor is valid for the time instant equal to the time corresponding to the middle of the window, the timestamp is calculated in that way. Explained process is being performed for each phase separately. For practical reasons, that is realized by using the Class named $Window$ and $SDFTWindow$. By calling three different instances of the class Window $ph1$,$ph2$,$ph3$ (pmu_rtd.py, lin 44-46) all methods are associated to three different phases.

After performing the set of testing experiments on the algorithm explained above, it has been noticed that the recursive DFT calculation causes numerical oscillations in case of off-nominal input signal frequencies. Additionally, due to the low accuracy of phasor and frequency estimations, the use of the mean filter is considered. The solution for the first issue has been found in reducing the window length from 80 to 20 samples. By doing that, each 4th input sample is consider in calculations, what is equivalent to the reduction of the sampling rate from 4kH to 1kHz. In that way, CPU load is decreased, enabling that the number of NON-recursive DFTs can be increased. On the other hand, in order to increase the accuracy of estimations, the mean filter, proposed in [12], is introduced. The filter with 1.5N order is applied to the frequency estimates, since the magnitude and phase estimations are directly based on calculated frequency. In order to apply the filter, the frequency should be estimated for each sample. Since it was previously estimated only in the reporting instances (10, 25 or 50 times per second), following modifications in the code are done. First of all, SDFT code is split in two methods: filtered frequency estimation and SDFT phasor estimation. The first method (frequency estimation) is being executed for each sample, while SDFT phasor estimation only at reporting instants. The filtering is performed on (2.5N +1) frequency estimates. The time tag was set at the middle of the data window, causing the measurement delay of only 1.25N samples. Once the filtered value of the frequency is obtained, and its time tag corresponds to the reporting rate, SDFT phasors are calculated and reported. The compared results for both, basic (b-SDFT) and enhanced (e-SDFT) version of the algorithm are presented in the section 7.4.

7.1. The calibration of the sensor

Due to the limited sampling accuracy of the ADC, the erroneous raw samples contribute the total vector error caused by the synchrophasor estimation technique. That is the reason why the calibration of the sensor has to be performed. In order to reduce the CPU load, the calibration module calibrates the estimated phasors at 50 Hz instead the raw samples at frequency of 4kHz. The calibration process expects the phasors in Cartesian complex form:

$$c_x = a_x + jb_x \tag{7.1}$$

where $a_x$and b_x are real and imaginary part of the estimated complex phasor respectively.

Following that, the calibration module takes the RMS value of the estimated phasor $z_x = \sqrt{a_x^2 + b_x^2}$ and performs correction on it. The calibration procedure involves searching for corresponding calibration points in the table (figure 7.4) and performing the linear interpolation on their equivalent RMS values. The calibration points are obtained in the RTDS laboratory at TU Delft by using Omicron Amplifier and True RMS Digital Multimeter Fluke shown in figure 7.1. The sensor is connected to the Omicron via coaxial cable, decreasing in that way electomagnetic interference (EMI) coming from the surrounding equipment (7.2). By using RSCAD, the software that interfaces the RTDS, the set of different voltage levels, from 0 to 250 Vrms, with the step of 10 V, is applied to the output of Omicron. For each voltage value, the calibration module is run in order to obtain the corresponding correction point. That value is then memorized in the calibration table, shown in figure 7.4. The same step is repeated for all applied rms voltage values.

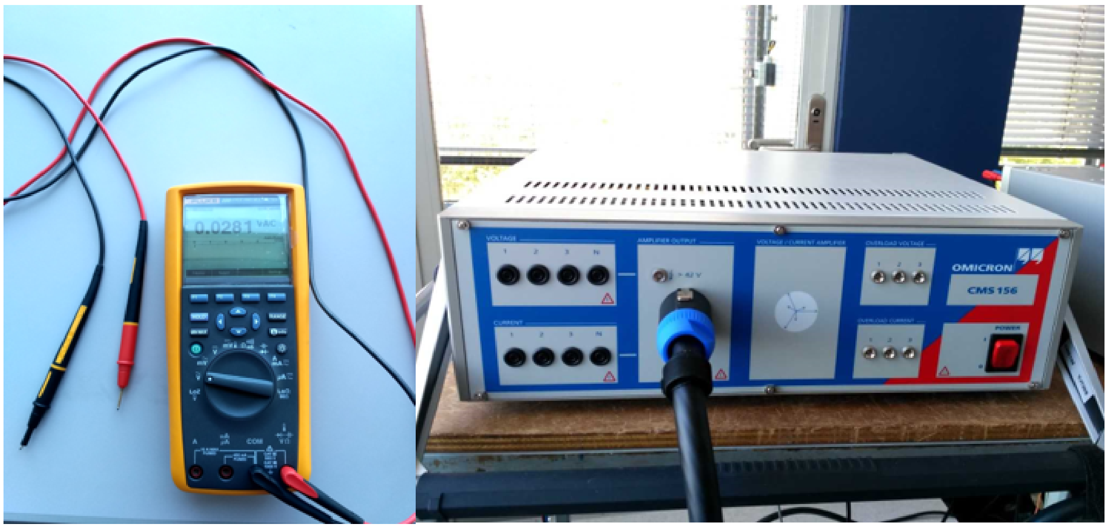


Figure 7.1: The instruments used for calibration

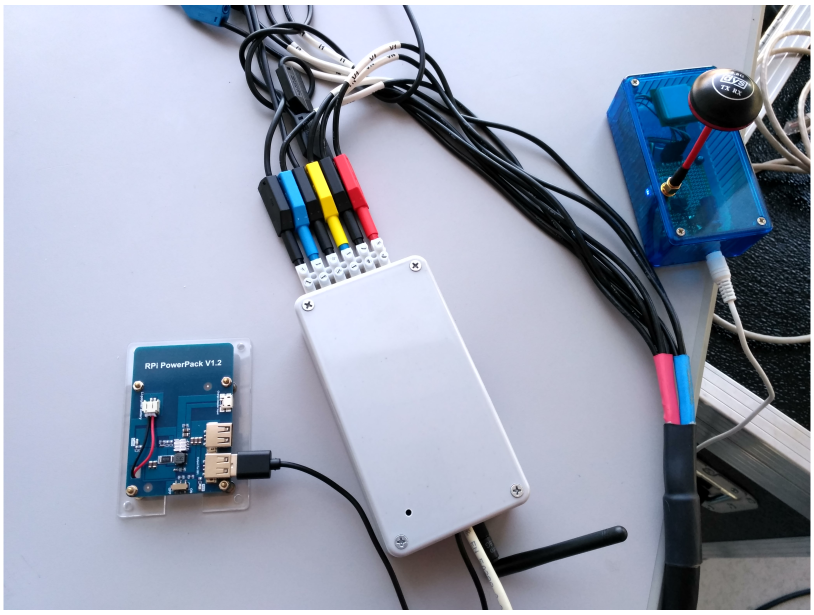


Figure 7.2: The sensor connections with Omicron

In the figure 7.3, the position of the calibration module with respect to other existing modules is shown. Namely, after raw samples are received from ADC and SDFT algorithm is performed, the estimated phasors are sent to the calibration module. The calibrated values are then routed towards communication module (section 7.3) that prepares the measurements for the transfer in the format compliant with the IEEE standard.

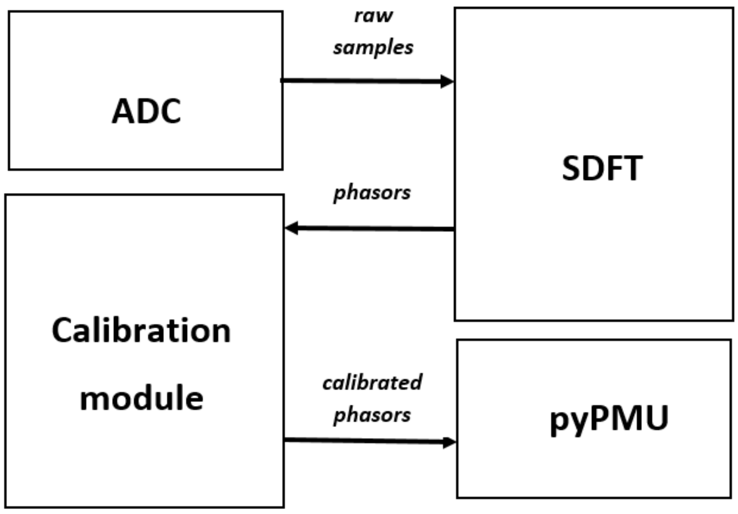
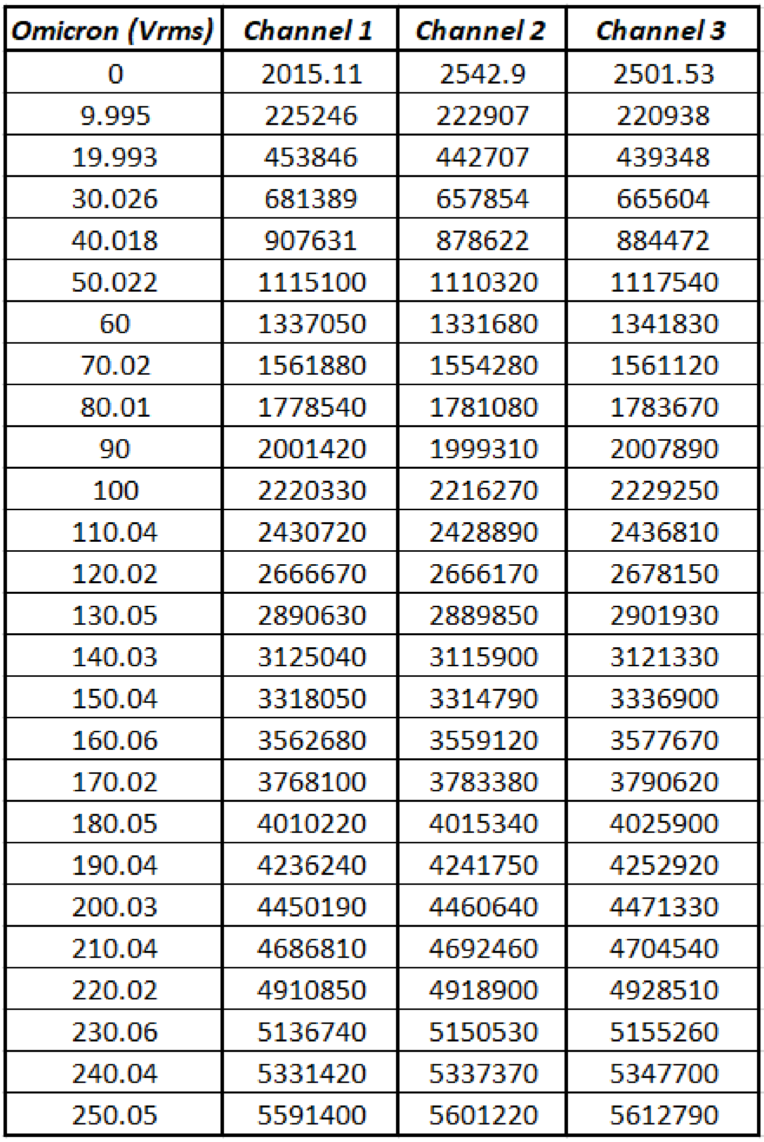


Figure 7.3: The correction of SDFT phasors



Omicron (Vrms)	Channel 1	Channel 2	Channel 3
0	2015.11	2542.9	2501.53
9.995	225246	222907	220938
19.993	453846	442707	439348
30.026	681389	657854	665604
40.018	907631	878622	884472
50.022	1115100	1110320	1117540
60	1337050	1331680	1341830
70.02	1561880	1554280	1561120
80.01	1778540	1781080	1783670
90	2001420	1999310	2007890
100	2220330	2216270	2229250
110.04	2430720	2428890	2436810
120.02	2666670	2666170	2678150
130.05	2890630	2889850	2901930
140.03	3125040	3115900	3121330
150.04	3318050	3314790	3336900
160.06	3562680	3559120	3577670
170.02	3768100	3783380	3790620
180.05	4010220	4015340	4025900
190.04	4236240	4241750	4252920
200.03	4450190	4460640	4471330
210.04	4686810	4692460	4704540
220.02	4910850	4918900	4928510
230.06	5136740	5150530	5155260
240.04	5331420	5337370	5347700
250.05	5591400	5601220	5612790

Figure 7.4: The calibration points for three channels

7.2. Electromagnetic Interference

After connecting the sensor to the function generator (source of the pure sine waves), it has been noticed that the input signal is significantly distorted for the voltage levels below 50 Vrms (7.5), while the sensor sensitivity increases as it reaches its nominal value of 230 Vrms.

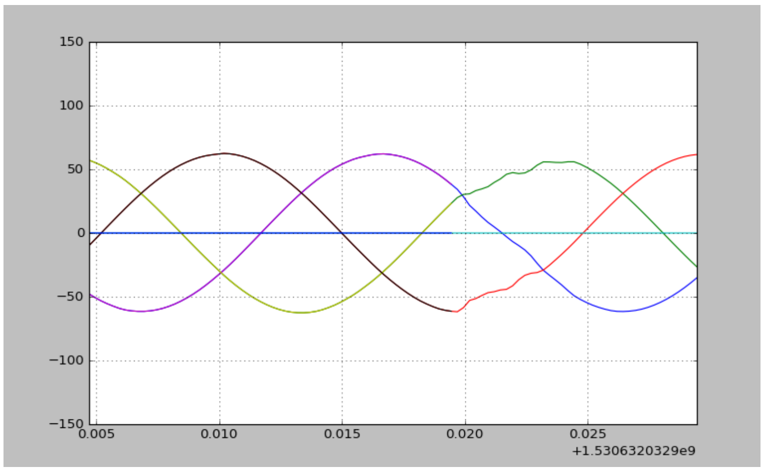


Figure 7.5: Distortion of the input sine wave

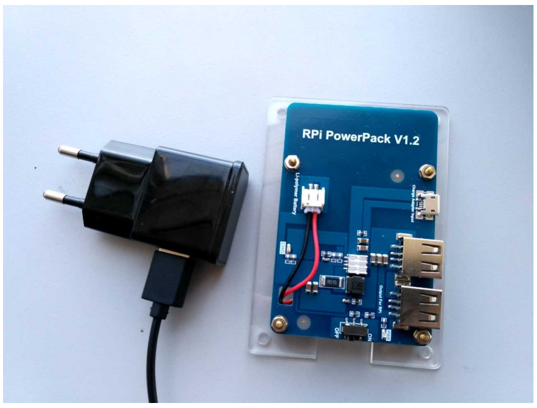


Figure 7.6: Adapter and battery power supply

The cause for the distortions has been found in conducted electromagnetic interference. Namely, the sensor is powered by using the power plug adapter that has insufficient filtering for the highly inductive conditions in the laboratory. The permanent solution for this issue could be adding extra EMI power supply filters in the new design of printed circuit board. However, in this project as a temporary solution, the adapter is replaced with the battery power supply and distortions are filtered out (figure 7.7). The adapter and battery supply of the sensor are shown in the figure 7.6

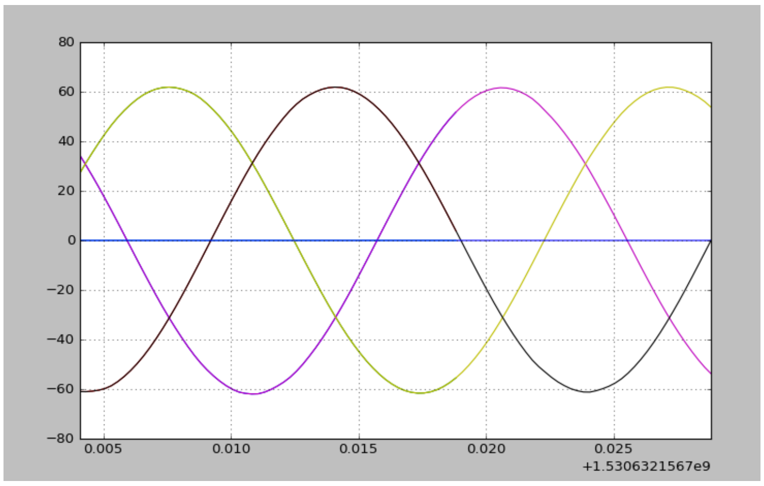


Figure 7.7: The input sine wave after replacing the adapter with the battery supply

7.3. PyPMU module

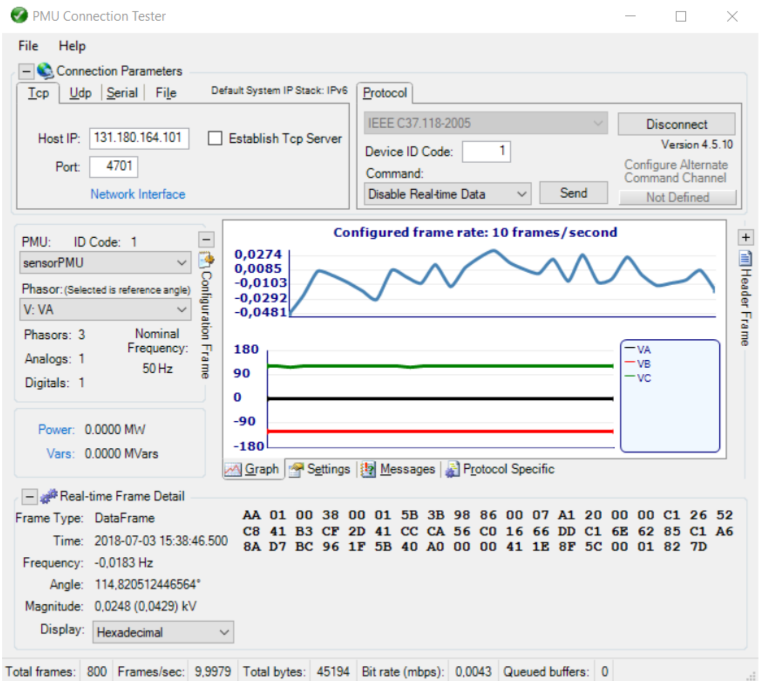
PyPMU synchrophasor module represents implementation of IEEE C37.118.2 standard for Data Transfer [20]. That is a communication protocol realization that enables the transfer of the estimated values from PMU to PDC. Due to the sensor processing power limitations, the pyPMU is run by the computer. The estimated frequency, phasors and corresponding timestamps are sent from sensor to the laptop via Ethernet port. Subsequently, the data frame with associated configuration frame is sent from pyPMU is sent to the PMU Connection Tester, the application used to validate, test and troubleshoot connections and data streams from phasor measurement units and graphically visualize the synchophasor estimates in real-time [21]. The data and configuration frame are shown in the following code. With the method *send_data*, the data frame is sent to specified IP address, while the configuration frame is defined by the configuration frame. The estimated values are sent from the sensor to the laptop by using the available TCP ports.Since one port carries information about the phasors, and another one about the frequency, they have to be synchronized, so that the correct pair of the frequency and corresponding phasor estimation are forwarded together to the pyPMU module. In the following code the method *send_data* is called with the estimated phasors, frequency and the timestamp as the arguments.

```

        pmu.send_data(phasors=[(self.package[msg[2]][0][7],self.package[msg[2]][0][8]),
                        (self.package[msg[2]][0][9],self.package[msg[2]][0][10]),
                        (self.package[msg[2]][0][11],self.package[msg[2]][0][12])],
                        analog=[9.91],
                        digital=[0x0001],
                        freq=self.package[msg[2]][1][8]-50,
                        dfreq=5,
                        stat=("ok", True, "timestamp", False, False, False, 0, "<10", 0),
                        soc=int(timestamp),
                        frasec =(int)((20000 * ((self.package[msg[2]][0][4])/80))%1000000)
                         )
```

In order to ensure that the forwarded values are sent with the original values and at the right time, the scaling factors, data format and rate of the phasor data transmission are set in the configuration frame.

In the figure 7.8 the PMU Connection Tester is shown. As an example, the estimated values are sent at the reporting rate of 10 frames/second. The PMU CT window is graphically showing the estimated frequency deviation and the phase estimation, as well as the numerical results of the magnitude estimation and the corresponding timestamp.



Default System IP Stack: IPv6

Host IP: 131.180.164.101

Port: 4701

Establish Tcp Server

IEEE C37.118-2005

Version 4.5.10

Device ID Code: 1

Command: Disable Real-time Data

PMU: ID Code: 1

sensorPMU

Phasor: (Selected is reference angle)

V: VA

Phasors: 3

Analogs: 1

Digitals: 1

Nominal Frequency: 50 Hz

Power: 0.0000 MW

Vars: 0.0000 MVars

Frame Type: DataFrame

Time: 2018-07-03 15:38:46.500

Frequency: -0,0183 Hz

Angle: 114,820512446564°

Magnitude: 0,0248 (0,0429) kV

Display: Hexadecimal

```
AA 01 00 38 00 01 5B 3B 98 86 00 07 A1 20 00 00 C1 26 52
C8 41 B3 CF 2D 41 CC CA 56 C0 16 66 DD C1 6E 62 85 C1 A6
8A D7 BC 96 1F 5B 40 A0 00 00 41 1E 8F 5C 00 01 82 7D
```

Total frames: 800 | Frames/sec: 9,9979 | Total bytes: 45194 | Bit rate (mbps): 0,0043 | Queued buffers: 0

Figure 7.8: PMU Connection Tester

The data transfer in the IEEE format is successfully established and verified on PMU Connection Tester running in the same computer where the sensor estimates were extracted to. However, the issues appeared while sending the data to the central computer in the laboratory. The data were being sent for a short period of time before the connection was interrupted. Each data frame was not of the same size, carrying the different number of messages. Consequently, the receiver was not able to sort out the messages and the connection failed. The problem was reported to the developers of the pyPMU. The investigation about the possible solutions is still going on.

7.4. The results of the real-time implementation

The testing of the PMU prototype is performed in RTDS laboratory at TU Delft. The testing conditions are obtained by using RSCAD and PMU Test Utility Tool for RTDS. Due to the communication issues between the sensor and the central computer (PDC) in the laboratory, complete Hardware-In-The-Loop is not realized and the results are collected directly from the sensor and saved in a .txt file. The file is imported to MATLAB in order to obtain the precise graphical representation.

7.4.1. Testing under the nominal frequency conditions

The testing of the real-time implementation of the SDFT algorithm has been started with applying a pure sine wave at 50Hz, with the magnitude of 100Vrms and phase of 0 radians. On the figure 7.9, the estimated magnitudes for all three phases are shown. It can be seen that the estimated values differ for each phase, and that the most precise estimate has been obtained for phase1, while phase2 and phase3 experience the offset of maximum 0.5 Vrms from the nominal value. The estimates are obtained by running the basic (b-SDFT) algorithm in the duration of 7s. The time in x-axes is given in UNIX format.

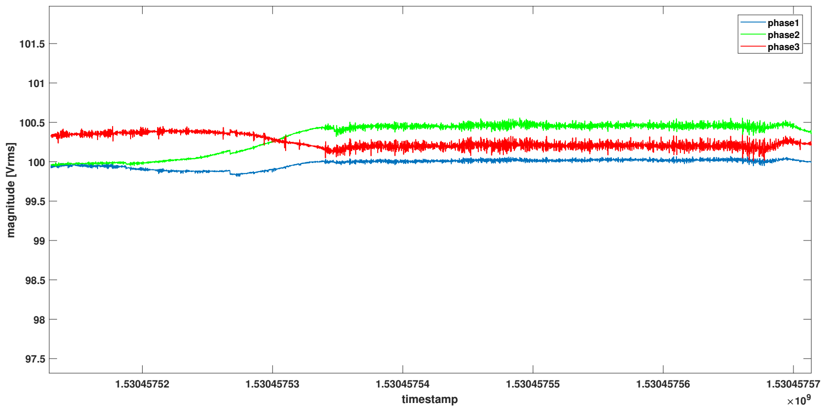


Figure 7.9: The algorithm performance during the steady-state conditions at 50Hz

After introducing the enhancements in the algorithm, by decreasing the number of recursive DFTs and introducing the mean filter, the accuracy of estimated magnitudes is improved. The maximum error is decreased to 0.05 Vrms, particularly present in the phase3 (7.10). The estimates are obtained for the time period of 12s. For the sake of simplicity, the time notation is changed from UNIX to human readable format.

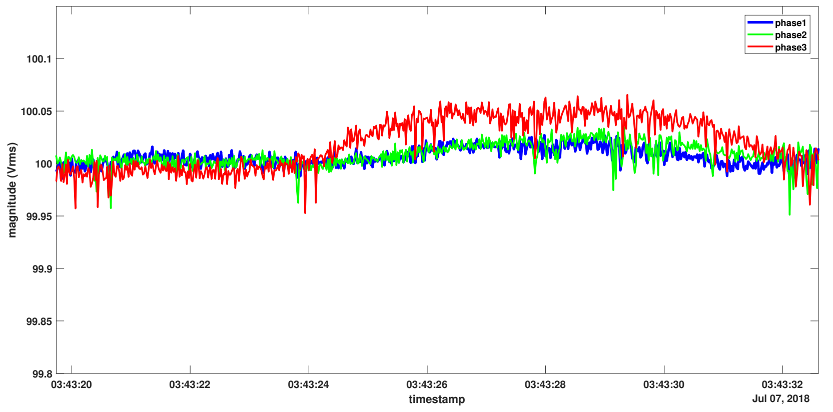


Figure 7.10: The enhanced algorithm performance during the steady-state conditions at 50Hz

On the figure 7.11, the estimated phases obtained with e-SDFT algorithm are plotted. As it can be seen, the phases are shifted by 120 degrees (by knowing that 120 degrees are equivalent to 2.0944 radians). The estimated phase depends on the moment when the DFT algorithm runs with the respect to the sampling of the signal. However, by choosing the phase1 as a reference with the initial phase 0, two other phases would get the values equal to -120 and -240 (120 degrees) respectively. The accuracy of phase estimation is not significantly changed by introducing the modifications in the algorithm.

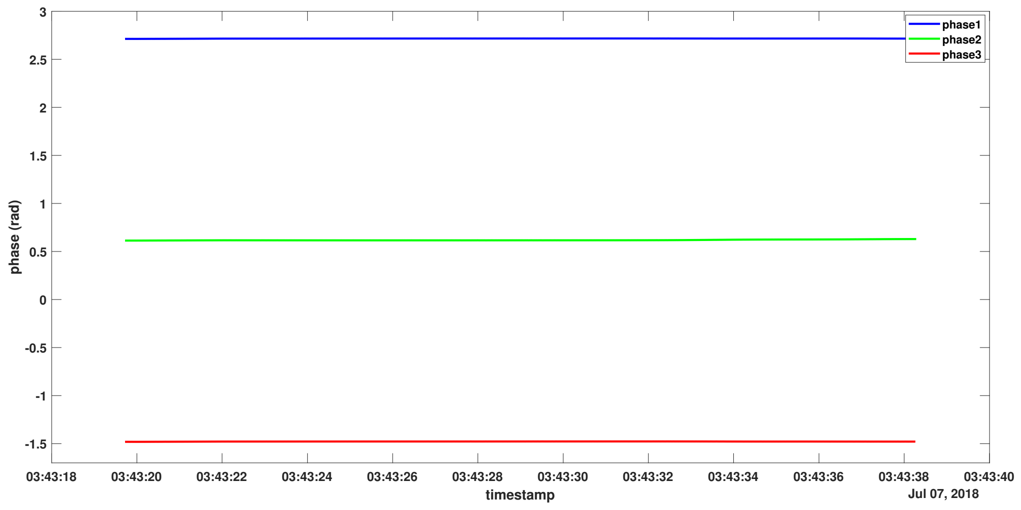


Figure 7.11: The phase estimation

The result of frequency estimation is given in the figure 7.12. Noticeably, enhanced algorithm has drastically improved the frequency estimation. The errors of almost 0.03 Hz caused by a basic algorithm are well suppressed by introducing the mean filter.

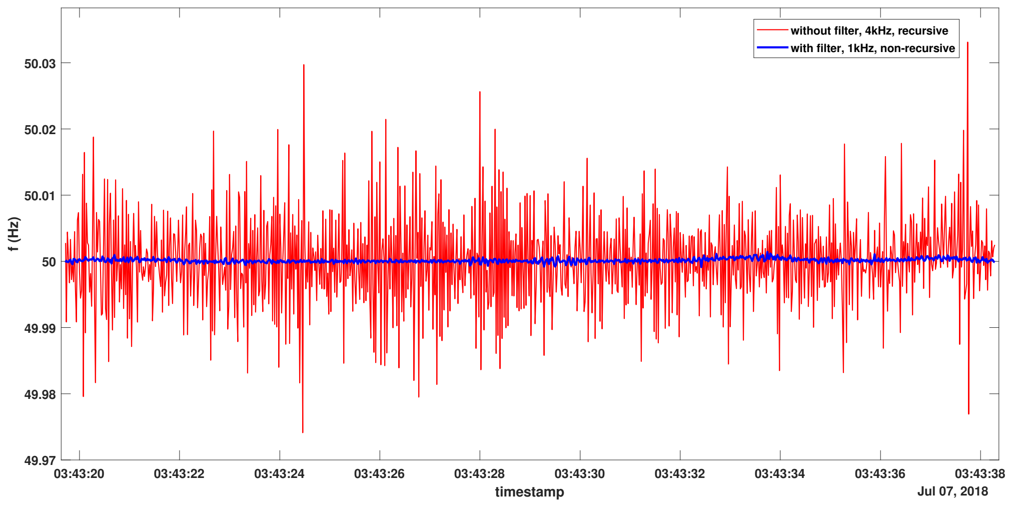


Figure 7.12: The comparison of the frequency estimation of the input signal at 50Hz

7.4.2. Testing under the off-nominal frequency conditions

Now, the testing conditions have been changed. The input signal frequency is increased from 50 Hz to 52 Hz. As a result of the frequency estimation based on a b-SDFT algorithm, the accuracy is decreased compared to the previous case at nominal frequency. The estimation experience the oscillations around 52 Hz, with the greatest deviation of 0.5 Hz (7.13). However, with the enhanced SDFT algorithm, the accuracy is improved and the deviations are attenuated. The maximum error occurring is only 0.03 Hz.

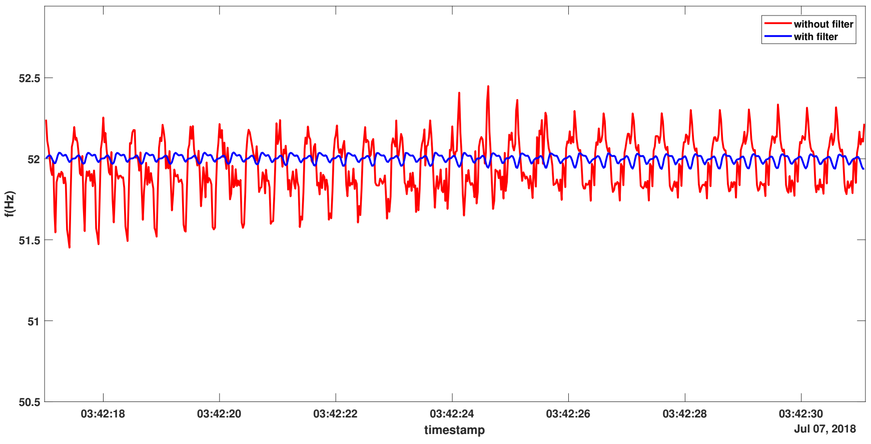


Figure 7.13: The comparison of the frequency estimation of the input signal at 52Hz

As a consequence of off-nominal input frequency, the magnitude estimation based on b-SDFT, experiences the numerical oscillations. As it can be seen from the figure 7.14, the values vary significantly from the reference value of 100 Vrms. As it was already mentioned, the oscillations are caused by unstable performance of recursive DFTs. By reducing the sampling rate to 1 kHz, it was ensured that CPU can handle NON-recursive DFT calculations. In that way, the numerical oscillations are avoided and accurate magnitude estimates obtained. The existing error is reduced from 8 Vrms to 0.3 Vrms.

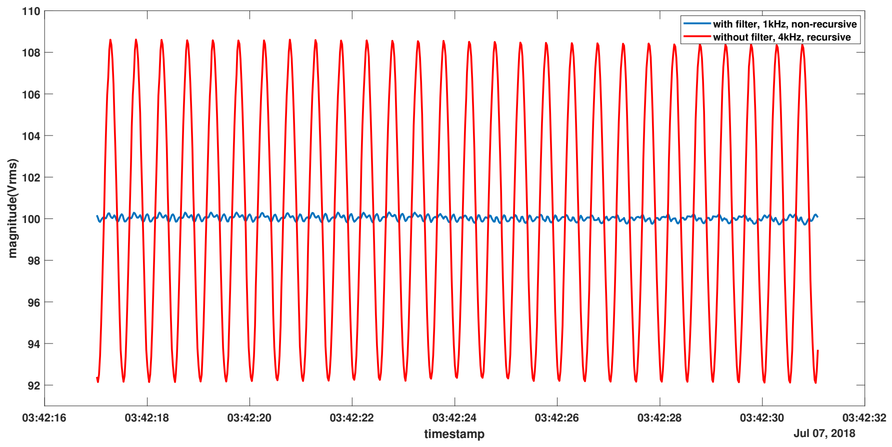


Figure 7.14: The comparison in magnitude estimation

As a result of the increased input signal frequency, the estimated phasors are rotating in the complex plane with the frequency equal to the deviation from the nominal value. For instance, the result of the phasor estimation of the input signal at 50.1 Hz is illustrated in the figure 7.15. Three phases with 120 degrees phase shift are rotating together at frequency of 0.1 Hz in counter-clockwise direction.

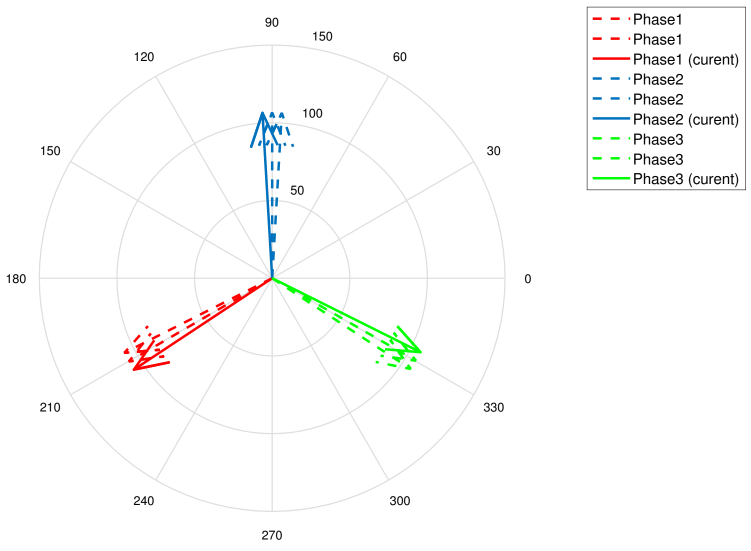


Figure 7.15: Counter-clockwise phase rotation at frequency Δf=0.1 Hz

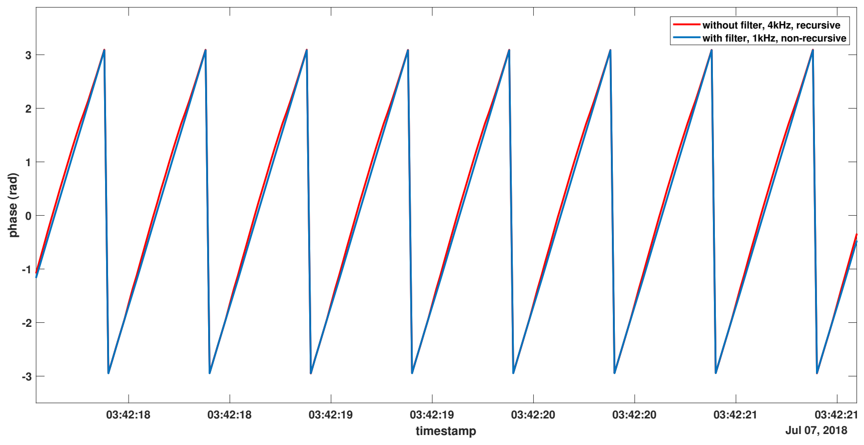


Figure 7.16: The phase estimation of the input signal at the frequency of 52 Hz

The results of the phase estimation for input signal at frequency of 52 is shown in the figure 7.16. Since the frequency is higher that nominal, the phase estimation has a positive slope. The figure shows the results of both algorithms. The slight improvement is obtained by running the enhanced version of SDFT.

7.4.3. The results of the synchrophasor estimation algorithm under the dynamic condition of the frequency ramp from 48Hz to 52Hz

The figures 7.17 and 7.18 show the results of the frequency estimation in case if the input signal frequency linearly varies from 48 Hz to 52 Hz for 4 s, for basic (b-SDFT) and enhanced algorithm (e-SDFT) respectively. As it can be seen, more accurate estimation is obtained by using the mean filter introduced with e-SDFT algorithm. The improvements are noticeable for all three parts of the signal at different frequencies: nominal start frequency, the frequency ramp and off-nominal stop frequency.

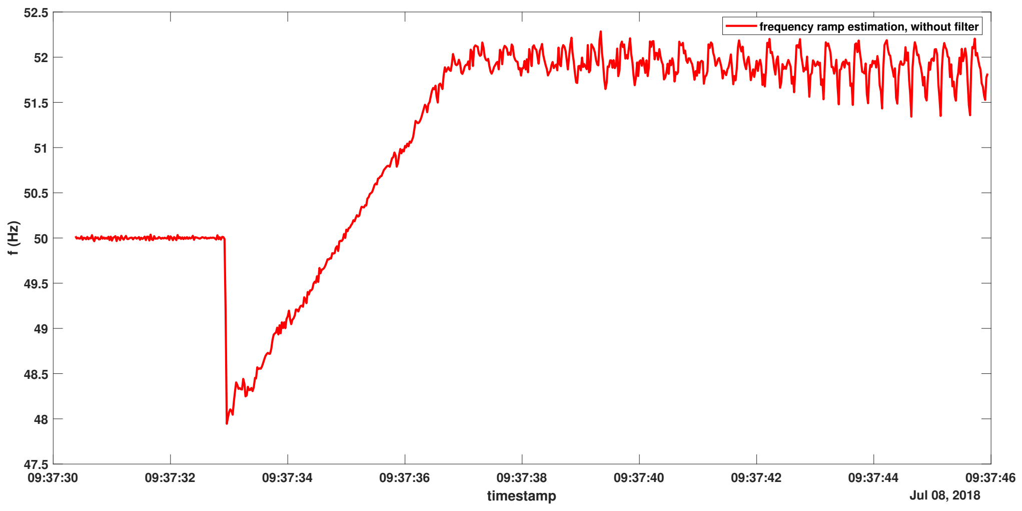


Figure 7.17: The performance of the basic algorithm in frequency ramp estimation

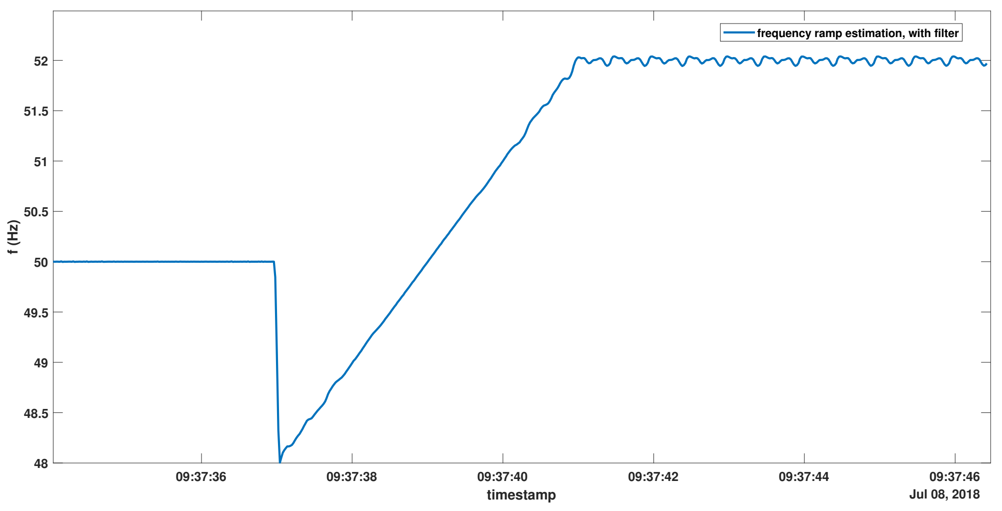


Figure 7.18: The performance of the enhanced algorithm in frequency ramp estimation

On the figure 7.19, the estimated magnitude during the frequency ramp is presented. Clearly, the accuracy is deteriorated during the transition period of 4 s. The deviations from nominal value vary from 0.05 to 0.5 Vrms.

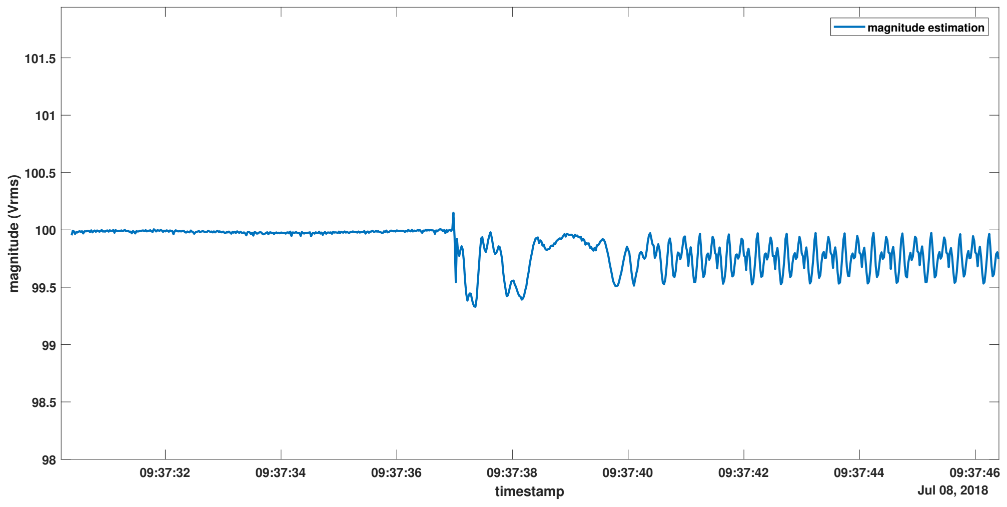


Figure 7.19: Magnitude estimation under the frequency ramp

7.4.4. The results of the synchrophasor estimation algorithm under the dinamic conditions - modulated input signal magnitude and phase

As it was already mentioned that the basic SDFT algorithm introduces the erroneous results in form of oscillations, the rest of the testing is performed only on the enhanced SDFT algorithm. In order to check the performance of the algorithm for the case when the input signal has a modulated magnitude, the magnitude is adjusted to vary with the modulation index of +/- 10% at frequency of 2 Hz. The result of the magnitude modulation in shown in the figure 7.24. During the period of 1 s, two magnitude cycles are occurring and magnitude value is varying from 90 to 110 Vrms.

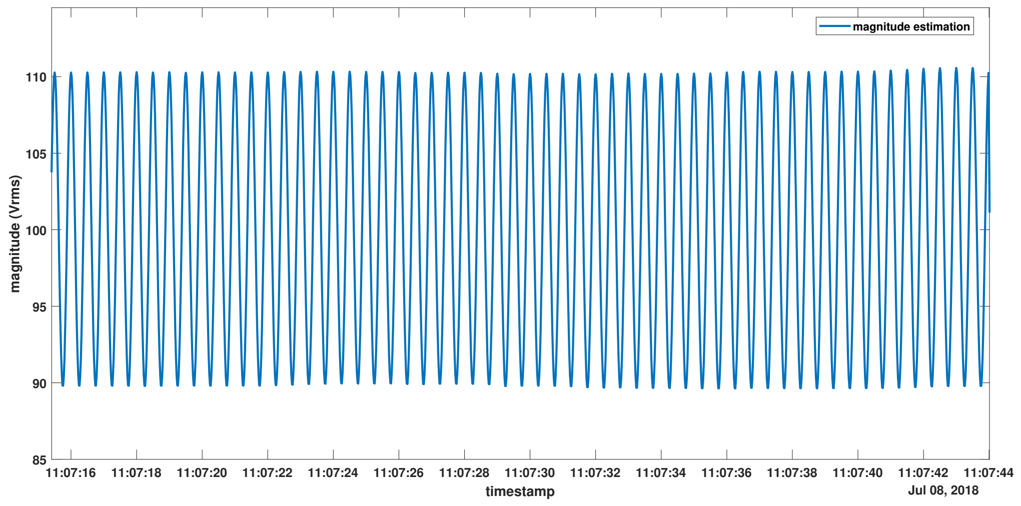


Figure 7.20: The magnitude estimation of the input signal with 10% modulation in magnitude

When it comes to the frequency estimation under the aforementioned conditions, the figure 7.21 indicates the presence of sinusoidal oscillations around the referent value of 50 Hz. The deviations are small with the maximum value of only 0.005 Hz.

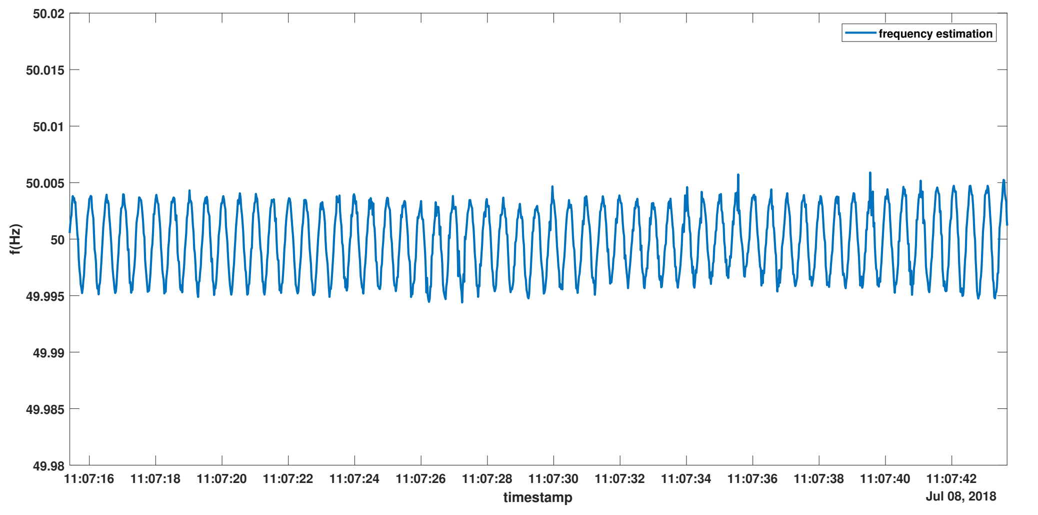


Figure 7.21: The frequency estimation of the input signal with 10% modulation in magnitude

After injecting the input signal with a modulated magnitude, the next test is aimed to check the performance of the e-SDFT algorithm under the dynamic condition caused by modulated input signal phase. The first plot 7.22 shows the results of the phase estimation. The phase is varying sinusoidally around 0 rad, with a slight deviation in magnitude.

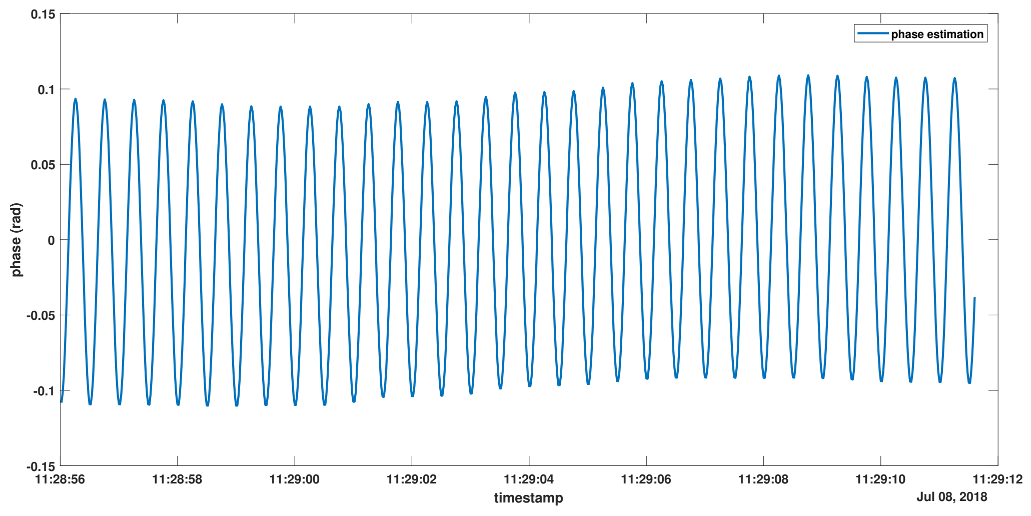


Figure 7.22: The phase estimation of the input signal with 10% modulation in phase

Similarly as for the frequency estimation during the magnitude modulation, the frequency estimation during the phase modulation experiences the oscillations around the reference value of 50 Hz. However, this time, the oscillations are more significant, reaching maximum value of 0.2 Hz (figure 7.23)

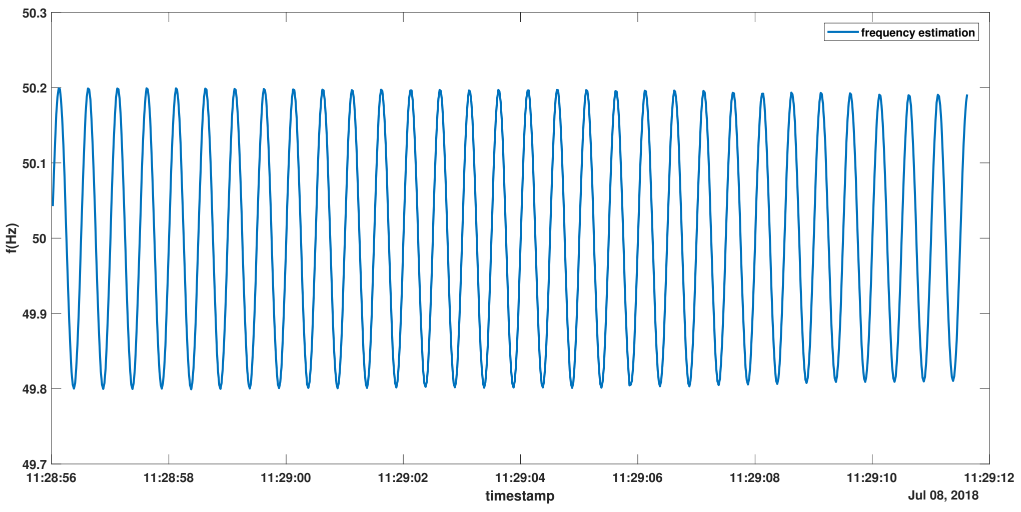


Figure 7.23: The frequency estimation of the input signal with 10% modulation in phase

7.4.5. The results of the synchrophasor estimation algorithm under the step in magnitude of the input signal

The final test represents the step in the magnitude of the input signal. The test starts with applying the magnitude step of 10 % and tracking the algorithm response. The figure 7.24 shows the result of estimation. The maximum error occurs at around 11:54:28, when the magnitude error reaches 0.9 Vrms.

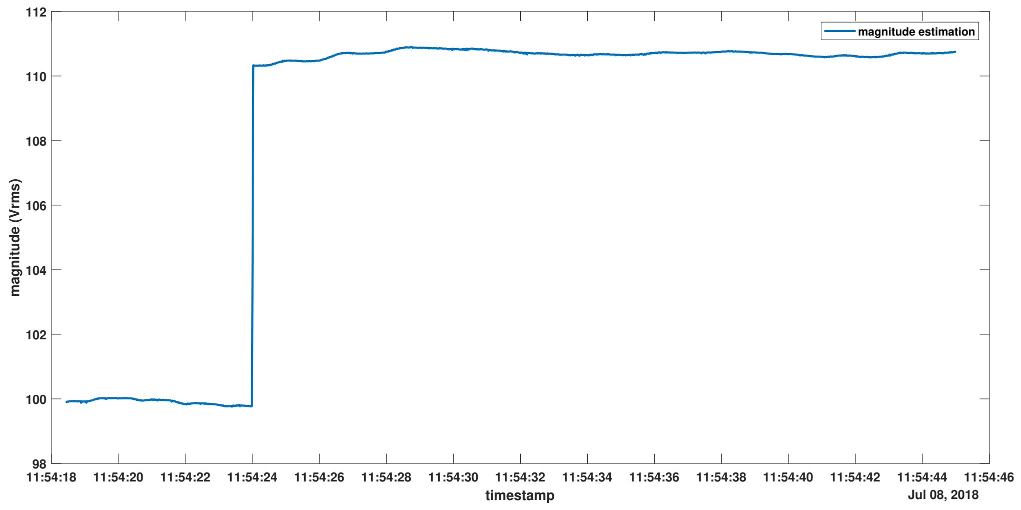


Figure 7.24: The magnitude estimation of the input signal with 10% step in magnitude

8 Conclusion

The thesis contributes in the development of the low-cost SDFT based Phasor Measurement Unit, that accurately estimates the magnitude, phase and frequency of the voltage waveforms coming from the electrical power grid. Throughout the thesis, the focus was on the optimized deployment of the computationally efficient algorithm for synchrophasor estimation to the low-cost voltage embedded sensor. The chosen algorithm is based on Smart Discrete Fourier Transform, due to the better accuracy that can be obtained under the off-nominal frequency conditions, the presence of higher harmonics and noise compared to the basic Discrete Fourier Transform. After the verification of the performance of SDFT technique in MATLAB environment, the code is developed for real-time applications in Python programming language. Implementation of the algorithm is optimized with respect to the hardware processing power. As a result of the optimization steps, two main versions of the code are obtained: b-SDFT (basic version) and e-SDFT (enhanced version). b-SDFT relies on 4kHz sampling rate, recursive DFTs and SDFT calculated at the reporting instants and it efficiently runs on sensor processing unit. On the other hand, more accurate e-SDFT is obtained by reducing the number of samples considered by one window, use of NON-recursive DFTs, sliding SDFTs and mean filtering of frequency estimates. Consequently, the main disadvantage of the recursive approach, the presence of numerical oscillation is avoided. Additionally, by increasing the number of SDFT calculations, the implementation of the mean filter is enabled. In that way, the accuracy of the estimation is significantly improved. Since the e-SDFT was computationally demanding for sensor, the processing power of the laptop is involved in estimation. In the latest stage, the e-SDFT is further optimized for sensor by decreasing the sampling rate of the ADC from 4kHz to 1 kHz. The testing of the prototype is performed with Real-Time Digital Simulator (RTDS). The test conditions are obtained by using PMU Utility in RSCAD, the software interfacing the RTDS. Due to the communications issues with PyPMU module for Data Transfer, the Hardware-in-the-Loop is not completely enabled and total vector error and measurement reporting latency are not calculated. With the further research, the reliable communication for data transfer and complete testing of the device is required. Currently, enhanced SDFT algorithm is being processed by low voltage sensor. The next step is implementation of the algorithm in the low current sensor. In that way, we will have two distributed synchronized sensors that are estimating current and voltage values separately. By sending estimates over IP to the aggregation hub, the power values will be calculated. Due to the low cost, the sensors could be massively deployed in distribution grid. The first application of SDFT Based PMUs will be power consumption monitoring at Technical University of Delft. The PMUs will be connected to the low voltage grid, and estimated measurements will be sent from the sensor to the monitoring unit that will display the values of voltage, current and frequency occurring in the grid in real-time.

A

A

A.0.1. Code in the script pmu_ rtd.py

```
#!/usr/bin/env python3
# Copyright (c) 2016-2018 Dennis Bijwaard (dennis@smartstatetechnology.nl)
#!/usr/bin/env python3
#from __future__ import print_function
import logging
import time
import json
import cbor
import DMN
import math
import signal
import sys
from os import path
#from Sample import Sample
from Window import Window
# Implement the default mpl key bindings
try:
    from configparser import ConfigParser
except:
    from ConfigParser import ConfigParser
from decimal import *
getcontext().prec = 4

from random import randint
if sys.version_info[0] < 3:
    import Queue as queue # for python2
else:
    import queue # for python>=3

class RTD_DSP():
    'DSP for calculating output from RTD values '
    def __init__(self, name):
        self.inputStarted=False
        self.dmn=None
        self.x=dict()
        self.y=dict()
        self.lasttime=time.time()
        self.index=0
        self.minFixTime=0
        self.delay=.001
        self.useTime=False
        self.running=True
        self.outputIDs=[]
        self.ph1 = Window(name ='Phase−1', id = 1)
        self.ph2 = Window(name ='Phase−2', id = 2)
        self.ph3 = Window(name ='Phase−3', id = 3)

    def scheduleQuit(self):
        logging.info("Stopping␣after␣closing␣window")
        self.running=False

    def add_output(self, rtdid, binding, outputIndex=0):
        logging.info("Adding␣output␣with␣outputIndex=%d"%outputIndex)
        self.outputIDs.append(rtdid)
        self.dmn.add_output(binding, outputIndex)

    def send_output(self, seq, ts, values, outputIndex=0):
        if len(self.outputIDs)>outputIndex:
            logger.debug("Sending␣to␣outputID[%d]=%d"%(outputIndex, self.outputIDs[outputIndex]))
            self.dmn.sendOutput(self.outputIDs[outputIndex], seq, ts, values, outputIndex)

    def updateInput(self, q):
        self.sampling_frequency=1000
        try:        #Try to check if there is data in the queue
            try:
                msg=q.get(timeout=self.delay)
                [tag, rtdidMain, rtd_items, rtd_len, seq, ts, interval]=msg[0:7]
#                print("q {} | {}|{}".format(len(msg) , q.qsize(), ts))
#                self.ph1.add_sample(msg[7::4])
#                self.ph2.add_sample(msg[8::4])
#                self.ph3.add_sample(msg[9::4])


                itemRange=range(0, rtd_items)
                xlist=[seq+j for j in itemRange]
#                self.ph1.add_seq(xlist)
#                self.ph2.add_seq(xlist)
#                self.ph3.add_seq(xlist)
                if seq==0:
                    ts=int(ts+0.5)

              # print("Time:", ts)
              # ts_gps=int(ts) + (seq)/self.sampling_frequency
                #Sts_gps=int(ts)


                tlist=[ts+interval*j for j in itemRange]

#                self.ph1.add_time(tlist)
#                self.ph2.add_time(tlist)
#                self.ph3.add_time(tlist)
                self.ph1.add_sample(msg[7::4][0::4], xlist[0::4], tlist[0::4])
                self.ph2.add_sample(msg[8::4][0::4], xlist[0::4], tlist[0::4])
                self.ph3.add_sample(msg[9::4][0::4], xlist[0::4], tlist[0::4])

                except Exception as e:
                    # print("i am in exception... yaay!! qdepth {}".format(  q.qsize()))
                     return self.running

                #logging.debug(msg)

            except Exception as e:
                logging.error("Error␣using␣input␣msg:␣%s"%(str(e)))
                return False
            return self.running

    def quit(self):
        self.dmn.quit()

class rtd_dmn(DMN.DMN):
    'Listening␣to␣a␣multitude␣of␣RTDs'
    def __init__(self, name, bindingRep):
        self.inputStarted=True
        self.last_ts=dict()

        super(rtd_dmn, self).__init__(name,bindingRep)

    def handle_command(self,req):
        '''
        default command handler, should be overriden in subclass
        @req request in json form
        '''
        if req['command']=="startDSP":
            resp="DSP␣started"
        else:
            logging.info("Received␣unknown␣command␣%s"%req)
            resp="unknown␣DSP␣command"
        return resp

    def handle_message(self,tag,msg):
        '''
        default message handler, should be overriden in subclass
        @tag name of this input
        @msg received message from publisher
        '''
        global q

        if not self.inputStarted:
            logging.warning("incoming␣message␣when␣not␣yet␣started")
            return

        length=len(msg)
        if length>=4:
            (rtdid,timing,rtd_items,rtd_len)=msg[0:4]
        else:
            logging.warning("not␣enough␣fields=%d␣in␣message",length)
            return
        rtd_size=rtd_items*rtd_len

        tagid="%s%d"%(tag,rtdid)
```

```python

        if isinstance(timing,int): # old behavior
            extraFields=length-5
            if timing<=0:
                usec=0
                seq=-timing
            else:
                usec=seq
                seq=None
            if extraFields>0:
                sec=msg[5]
            else:
                sec=None
            if extraFields>1:
                usec=msg[6]
        else: # timing array
            seq=None
            sec=None
            usec=0
            try: # try setting until exception, (u)sec may not be available
                seq=timing[0]
                sec=timing[1]
                usec=timing[2]
            except:
                pass

        if seq!=None:
            interval=1 # default to 1 sequence number interval
            if rtd_items>1:
                seq=seq-rtd_items+1
        if sec!=None:
            ts=sec+usec/1000000 # use timestamp
            default_interval=.0001 # default to low interval until last_ts is available
            interval=default_interval
            if rtd_items>1:
                #logging.debug("rtdid=%d, msg ts=%f,items=%d",rtdid,ts,rtd_items)
                try:
                    last_ts=self.last_ts[tagid]
                    interval=(ts-last_ts)/rtd_items
                    if interval>1: # probably missed a bunch of samples
                        interval=default_interval
                        last_ts=ts-interval*rtd_items
                    else:
                        if interval<0: # time goes backwards
                            logging.warn("rtdid=%s, msg ts=%f,interval=%f",tagid,ts,interval)
                            interval=default_interval
                except KeyError: # last_ts is not yet available
                    last_ts=ts-interval*rtd_items
                self.last_ts[tagid]=ts
                ts=last_ts+interval
        else:
            ts=None

        value=msg[4]
        if isinstance(value,list):
            q.put([tag,rtdid,rtd_items,rtd_len,seq,ts,interval]+value)
        else:
            q.put([tag,rtdid,rtd_items,rtd_len,seq,ts,interval,value])

    def sendOutput(self, rtdid, seq, ts, values,outputIndex=0):
        '''
        send values as DSP output (non-aggregated, rtd_items=1)
        @param seq the seqence number for the result, or None when sequence is not known
        @param ts  the timestamp for the result
        @param values an array of values, or single value containing the result
        @param outputIndex the index of the output
        '''
        sec=int(ts)
        usec=int((ts-sec)*1000000)
        msg=[rtdid,[seq,sec,usec],1,len(values),values]
        self.send_output(msg,outputIndex)

    def quit(self):
        self.msgr.send_command(self.bindingReq, '{␣"command":"stop"␣}')

def dsp_realtime(dsp_name):
    #dsp=RTD_DSP(dsp_name,"ipc:///tmp/dsp%d"%(randint(0,100)))
    dsp=RTD_DSP(dsp_name)
    variant="sdf"
    # read configuration
    configfile=path.join(path.dirname(path.realpath(__file__)),"rtd.properties")
    defaults={ "sensors":["sensor1"],"outputs":[] }
    config=ConfigParser(defaults)
    config.read(configfile)
    dsp.dmn=rtd_dmn(dsp_name,"inproc://tmp/%s"%(dsp_name))

    sensors=json.loads(config.get("defaults","sensors"))
    # listen to realtime data
    for sensor in sensors:
        ports=json.loads(config.get(sensor,variant+"_ports"))
        host=config.get(sensor,"tcp_host")
        try:
            tag=config.get(sensor,"tag")
        except Exception as e:
            logging.error("Error␣config␣msg:␣%s"%(str(e)))
            tag=host
        for port in ports:
            dsp.dmn.add_subscription(tag,"tcp://%s:%d"%(host,port))

    outputs=json.loads(config.get("defaults","outputs"))
    outputIndex=0
    for output in outputs:
        binding=config.get(output,variant+"_binding")
        rtdid=config.get(output,variant+"_rtdid")
        if isinstance(rtdid,str):
            rtdid=int(rtdid)
        logging.info("Adding␣binding␣%s␣to␣output␣with␣rtdid␣%d"%(binding,rtdid))
        dsp.add_output(rtdid,binding,outputIndex)
        outputIndex+=1

    return dsp

```

```
def dsp_all():
    global q,running,stopping
    stopping=False
    signal.signal(signal.SIGINT, signal_handler)
    q = queue.Queue()
    dsp=dsp_realtime('MyDSP')
    running=True
    thread = False
    if( thread):
        dsp.ph1.start()
        time.sleep(0.1)
        dsp.ph2.start()
        time.sleep(0.1)
        dsp.ph3.start()
        time.sleep(0.1)

    #running=dsp.updateInput(q)
    #running=dsp.updateInput(q)
    while (running):
        running=dsp.updateInput(q)
    #    print("TIME:", dsp.ph1.input_time)
     #   print("SEQ:", dsp.ph1.input_seq)
       # print("PHASE 1:", dsp.ph1.input_sample)

#          #start = time.time()
        if( not thread):
            for i in range(0,10):
                sending =dsp.ph1.new_test()
                sending =dsp.ph2.new_test()
                sending =dsp.ph3.new_test()
                if(sending and dsp.ph1.sdft_window.input_sample_seq[25]%(80)==0):
                    #seq,ts,values,outputIndex=0
                   #print("sending")
                    dsp.ph1.sdft_window.calculateResult(1)
                    dsp.ph2.sdft_window.calculateResult(2)
                    dsp.ph3.sdft_window.calculateResult(3)
                    temp= dsp.ph2.sdft_window.result + [1.0,1.0]
                    print("Sending␣Values:{},{},{}".format(
                    dsp.ph1.sdft_window.input_sample_time[25],
                    dsp.ph1.sdft_window.input_sample_seq[25], temp ))
                    print("Sending␣Freq:{},{},{}".format(
                    dsp.ph1.sdft_window.input_sample_time[25],
                    dsp.ph1.sdft_window.input_sample_seq[25],
                    dsp.ph1.sdft_window.freq ))

                    #print("".format())
                    dsp.send_output(ts = dsp.ph1.sdft_window.input_sample_time[25],
                    seq = dsp.ph1.sdft_window.input_sample_seq[25],
                    values  = temp,outputIndex = 0)
                    dsp.send_output(ts = dsp.ph1.sdft_window.input_sample_time[25],
                    seq = dsp.ph1.sdft_window.input_sample_seq[25],
                    values  = dsp.ph1.sdft_window.freq, outputIndex = 1)


        #dsp.ph1.test_recursive()

```

```
            #dsp.ph2.test()
            #dsp.ph3.test()
            #end = time.time()
            #print("Time passed: ",end - start)
            #print(dsp.window.input_sample)
            #time.sleep(1)
            if stopping:
                logger.debug('Trying␣to␣stop␣after␣CTRL-C!')
                Window.exit = True
                running=False

    logger.debug('Trying␣to␣quit␣after␣CTRL-C!')
    dsp.quit()
    logger.debug('Quited␣after␣CTRL-C!')

def signal_handler(signal, frame):
    global stopping
    logger.info('You␣pressed␣Ctrl+C!')
    stopping=True

if __name__ == "__main__":
    logfile='session-dsp.log'
    logger = logging.getLogger('rtd_dsp')
    logging.basicConfig(level=logging.INFO,
                        format='%(asctime)s.%(msecs)03d␣%(levelname)-8s␣%(message)s',
                        datefmt='%Y-%m-%d,%H:%M:%S',
                        filename=logfile,
                        filemode='w')
    stderrLogger=logging.StreamHandler()
    stderrLogger.setLevel(logging.INFO)
    stderrLogger.setFormatter(logging.Formatter(logging.BASIC_FORMAT))
    logger.addHandler(stderrLogger)

    dsp_all()

    logging.info("Stopped,␣running=%d"%(running))
```

A.0.2. Code in the script Window.py

```
import numpy as np
from SDFTWindow import SDFTWindow
import time
import threading
from decimal import *



class Window(threading.Thread):
    """
    Constants definition
    """
    WINDOW_LENGTH = 20
    MAX_RECURSION =0
    MAX_LEN =40
    sampling_frequency=1000
    n=range(0,WINDOW_LENGTH)
    Factor_re=(1/WINDOW_LENGTH)*np.cos(np.multiply((-2*np.pi)/WINDOW_LENGTH,n))
    Factor_im=(1/WINDOW_LENGTH)*np.sin(np.multiply((-2*np.pi)/WINDOW_LENGTH,n))


    m=range(0,MAX_RECURSION)
    Factor_re_rec=np.sqrt(2)*(1/WINDOW_LENGTH)*np.cos(np.multiply((-2*np.pi)/WINDOW_LENGTH,m))
    Factor_im_rec=np.sqrt(2)*(1/WINDOW_LENGTH)*np.sin(np.multiply((-2*np.pi)/WINDOW_LENGTH,m))
    Factor_rec=Factor_re_rec+np.multiply(1j,Factor_im_rec)
    exit = False
    c1=2*np.pi/WINDOW_LENGTH
    c2=np.sqrt(2)*(1/WINDOW_LENGTH);
    Factor_1=(np.cos(c1)+1j*np.sin(c1))

    def __init__(self, window_position = 0, input_sample = [], input_time = [],
    input_seq = [], window_length = 20 , max_len = 40, block = 1,
    block_t=1, block_s=1, name = 'Phase', id = 0):
        threading.Thread.__init__(self)
        self.name = name
        self.recursive_count = 0
        self.id = id
        self.window_position = 20
        self.input_sample = input_sample
        self.input_time = input_time
        self.input_seq = input_seq
        self.sample_count= 0
        self.sample_count_t= 0
        self.sample_count_s= 0
        self.block = block
        self.block_t=block_t
        self.block_s=block_s
        self.sdft_window = SDFTWindow()
        self.doDFT = False
        self.runSDFT= False
        self.reset = 0
        self.non_recursive_count=0
        self.dft_count=0
        self.input_sample =[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        self.input_sample = self.input_sample[0:40]
        self.input_time=[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        self.input_time = self.input_time[0:40]
        self.input_seq=[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        self.input_seq = self.input_seq[0:40]

def new_test(self):
        sending = False;
        if(   ((self.reset*self.MAX_LEN) + self.sample_count) > self.window_position ):

            if(self.recursive_count == 0 ):

                self.runDFT()
                self.dft_count+=1
            else:
                self.runDFT_recursive()
                self.dft_count+=1
            #print("dft count : {} for {}".format(self.dft_count, self.name))
            #if (self.input_seq[self.window_position-10]%(self.WINDOW_LENGTH)==0):
                #print("{} window {} phase".format(self.window_position , self.idA
            self.sdft_window.runSDFT(self.id)
               # print(self.input_seq[self.window_position-40])

            if(self.id == 3):
               # self.sdft_window.print_result()
                sending = True

            self.dft_count%=self.MAX_RECURSION+1
            self.recursive_count += 1
            self.recursive_count %= self.MAX_RECURSION+1
            self.window_position += 1
            if(self.window_position == self.MAX_LEN):
                self.window_position = 0 # self.window_position %= self.MAX_LEN
                self.reset = 0
            return sending
def add_sample(self , arr, arr1, arr2):

        if(self.block == 1):
            #print("hello1 "+ str(arr))
            self.input_sample = arr + self.input_sample[ 10:self.MAX_LEN]
            self.input_seq = arr1 + self.input_seq[ 10:self.MAX_LEN]
            self.input_time = arr2 + self.input_time[ 10:self.MAX_LEN]

            self.block = 2
            self.sample_count += 10

        elif(self.block == 2):
            #print("hello2 "+ str(arr))
            self.block = 3
            self.input_sample = self.input_sample[0:10]+ arr + self.input_sample[20:self.MAX_LEN]
            self.input_seq = self.input_seq[0:10]+ arr1 + self.input_seq[20:self.MAX_LEN]
            self.input_time = self.input_time[0:10]+ arr2 + self.input_time[20:self.MAX_LEN]

            self.sample_count += 10

        elif(self.block == 3):
                #print("hello3 "+ str(arr))
            self.block = 4
            self.input_sample = self.input_sample[0:20] + arr +
            self.input_sample[30:self.MAX_LEN]
            self.input_seq = self.input_seq[0:20] + arr1 +
            self.input_seq[30:self.MAX_LEN]
            self.input_time = self.input_time[0:20] + arr2 +
            self.input_time[30:self.MAX_LEN]

            self.sample_count += 10

        elif(self.block == 4):
                #print("hello3 "+ str(arr))
            self.block = 1
            self.input_sample = self.input_sample[0:30] + arr
            self.input_seq = self.input_seq[0:30] + arr1
            self.input_time = self.input_time[0:30] + arr2

            self.reset = 1
            self.sample_count = 0

        else:
             print("Error")
    def runDFT(self):

       # print("running dft")
        bin_1_phase_re = 0
        bin_1_phase_im = 0
        for i in range(0,20):


            bin_1_phase_re = bin_1_phase_re +
            (self.input_sample[(self.window_position - 20 + i)% self.MAX_LEN ]
            * (self.Factor_re[i]))bin_1_phase_im = bin_1_phase_im +
            (self.input_sample[(self.window_position - 20 + i)% self.MAX_LEN ]
            * (self.Factor_im[i]))

        X_k_phase_re= 1.414 *bin_1_phase_re
        X_k_phase_im= 1.414 *bin_1_phase_im


        X_k_phase=X_k_phase_re+1j*X_k_phase_im
        self.sdft_window.input_sample= self.sdft_window.input_sample[1:50] + [X_k_phase]


        time_stamp=self.input_time[self.window_position - 10]
        self.sdft_window.input_sample_time=self.sdft_window.input_sample_time[1:50]
        +[time_stamp]
        middle_seq=self.input_seq[self.window_position - 10]
        self.sdft_window.input_sample_seq=self.sdft_window.input_sample_seq[1:50] +
        [middle_seq]

def runDFT_recursive(self):

        X_new=(self.sdft_window.input_sample[-1] +
        self.c2*(self.input_sample[(self.window_position)%self.MAX_LEN]-
        self.input_sample[self.window_position-20]))*self.Factor_1
        self.sdft_window.input_sample= self.sdft_window.input_sample[1:50] + [X_new]


        time_stamp=self.input_time[self.window_position - 10]
        self.sdft_window.input_sample_time=self.sdft_window.input_sample_time[1:50] +[time_stamp]
        middle_seq=self.input_seq[self.window_position - 10]
        self.sdft_window.input_sample_seq=self.sdft_window.input_sample_seq[1:50] +[middle_seq]
```

A.0.3. Code in the script SDFTWindow.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May  8 20:44:22 2018

@author: isidora
"""

# -*- coding: utf-8 -*-
"""
Created on Mon Apr 23 11:40:54 2018

@author: isidora
"""

import numpy as np
#import time

class SDFTWindow(object):
    """
    Constants definition
    """
    WINDOW_LENGTH = 3
    MAX_LEN = 3
    Fs=1000
    f=50
    N=20
    c1=Fs/(2*np.pi)
    c2=(np.pi)/Fs
    c3=np.e
    result=[-1,-1,-1,-1,-1,-1]
    freq = [-1,-1,-1]


    def __init__(self, window_position = 0, input_sample = [], window_length = 3 ,
    max_len = 3, n_phase = 3, result = [], input_sample_time=[],input_sample_seq=[]):
        self.window_position = window_position
        self.freq_filter_window = [];
#         self.input_sample_re = [-1,-1,-1]
#         self.input_sample_im = [-1,-1,-1]
        #self.result = result
        self.i = 2
        self.input_sample = [-1,-1, -1, -1, -1,-1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1]
        self.input_sample_time=[-1,-1, -1, -1, -1,-1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
        self.input_sample_seq=[-1,-1, -1, -1, -1,-1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

        #self.result = [-1]

    def getAverage(self, arr):
        sum =0;
        for i in range(0, len(arr)):
            sum += arr[i];

        return sum/(len(arr))

    def runSDFT(self, id):

#         if(self.i >0):
#             self.i -= 1
#             return
        X_k_arr=self.input_sample[len(self.input_sample)-3 : len(self.input_sample)]
        w=(X_k_arr[0]+X_k_arr[2])/X_k_arr[1]
        self.freq_filter_window = [(self.c1)*np.arccos(np.real(w/2))]
        + self.freq_filter_window[0:50]




    def calculateResult(self, id=0):
        f_es = self.getAverage(self.freq_filter_window)
        X_k_arr=self.input_sample[25:28]
        f_delta=f_es-self.f
        a1=np.cos(2*self.c2*(self.f+f_delta))+1j*np.sin(2*self.c2*(self.f+f_delta))
        #Ar=((a1*a1)*X_k_arr[0]-a1*X_k_arr[1])/(a1*a1-1)
        Ar=(X_k_arr[1]*a1-X_k_arr[0])/(a1*a1-1)
        X_est=(abs(Ar))*((self.N*np.sin(self.c2*f_delta))/(np.sin(self.N*self.c2*f_delta)))
        phi_es=np.angle(Ar)-(self.c2*(self.N-1)*f_delta)

        X_meas=X_est*pow(self.c3,1j*phi_es)
        self.result[2*(id-1)] =  np.real(X_meas).tolist()
        self.result[2*(id-1)+1] = np.imag(X_meas).tolist()
        self.freq[id-1] = f_es

        #self.result=self.result1.tolist()


        #end = time.time()
        #print("SDFT time {}".format(end - start))

    def print_result(self):
        #print("SDFT phasors are:",self.result)
        #print("estimated frequency is:", self.freq)
        #print("{} stamp".format(self.input_sample_time[2]))
       # print("Time stamp:",self.input_sample_time[2])
        #print("Sequence:", self.input_sample_seq[2])
        #print("Magnitude:",abs(self.result[0]+1j*self.result[1]))
        print("Output:␣{}|{}|{}|{}|{}|{}␣".format(self.input_sample_time[25],
        self.input_sample_seq[25],self.freq[0],abs(self.result[0]+1j*self.result[1]),
        abs(self.result[2]+1j*self.result[3]),abs(self.result[4]+1j*self.result[5])))
```

A.0.4. Code in the script rtd_pyPMU.py

```
#!/usr/bin/env python3
# Copyright (c) 2016-2018 Dennis Bijwaard (dennis@smartstatetechnology.nl)
# Distributed under the Boost Software License, Version 1.0.
# (See https://www.boost.org/LICENSE_1_0.txt)
#from __future__ import print_function
from synchrophasor.frame import ConfigFrame2
from synchrophasor.frame import CommonFrame
from synchrophasor.pmu import Pmu
from time import time
from time import sleep
import logging
import time
import json
import cbor
import DMN
import math
import signal
import sys
from os import path
import timeit
# Implement the default mpl key bindings
try:
    from configparser import ConfigParser
except:
    from ConfigParser import ConfigParser

from random import randint
if sys.version_info[0] < 3:
    import Queue as queue # for python2
else:
    import queue # for python>=3

class RTD_DSP():
    'DSP␣for␣calculating␣output␣from␣RTD␣values␣'
    def __init__(self, name):
        self.inputStarted=False
        self.dmn=None
        self.x=dict()
        self.y=dict()
        self.lasttime=time.time()
        self.index=0
        self.minFixTime=0
        self.delay=.001
        self.useTime=False
        self.running=True
        self.outputIDs=[]
        self.package = []
        for i in range(0,4000):
            self.package.append([[],[],0])

    def scheduleQuit(self):
        logging.info("Stopping␣after␣closing␣window")
        self.running=False

    def add_output(self,rtdid,binding,outputIndex=0):
        logging.info("Adding␣output␣with␣outputIndex=%d"%outputIndex)
        self.outputIDs.append(rtdid)
        self.dmn.add_output(binding,outputIndex)

    def send_output(self,seq,ts,values,outputIndex=0):
        if len(self.outputIDs)>outputIndex:
            logger.debug("Sending to outputID[%d]=%d"%(outputIndex,self.outputIDs[outputIndex]))
            self.dmn.sendOutput(self.outputIDs[outputIndex],seq,ts,values,outputIndex)
     #def updateInput(self,q,pmu):
    def updateInput(self,q,pmu):
        self.sampling_frequency=1000
        try:        #Try to check if there is data in the queue
            try:
                msg=q.get(timeout=self.delay)
            except Exception as e:
                return self.running
            [tag,rtdidMain,rtd_items,rtd_len,seq,ts,interval]=msg[0:7]
            #msg= [tag,rtdidMain,rtd_items,rtd_len,seq,int(ts+.5) + (seq)*0.00025,interval]
            #ts_gps=int(ts+.5) + (seq)*0.00025
            #print("TIME STAMP:",ts_gps)
            #print("hello1")
            #print(msg)
           # print("msg:",msg[1])
            #logging.debug(msg)
            if(msg[1] == 1014):
                #print('1003:{}'.format(self.package[msg[2]][2]))
                self.package[msg[2]][0] = msg
                self.package[msg[2]][2]+=1
            else:
                #print('2001:{}'.format(self.package[msg[2]][2]))
                self.package[msg[2]][1] = msg
                self.package[msg[2]][2]+=1

            if(self.package[msg[2]][2] == 2):
                self.package[msg[2]][2]=0;

                if pmu.clients:
                        #sleep(1/100)
                        #timestamp=(int)(((dsp.ph1.sdft_window.input_sample_seq[1]+1))/80)*20
#                         if self.package[msg[2]][0][4] ==0:
#                             timestamp=int(self.package[msg[2]][0][5]+0.5)
#                         else:
#                             timestamp=int(self.package[msg[2]][0][5])

                        timestamp=int(self.package[msg[2]][0][5])

                        pmu.send_data(phasors=[(self.package[msg[2]][0][7],
                        self.package[msg[2]][0][8]),(self.package[msg[2]][0][9],
                        self.package[msg[2]][0][10]),
                        (self.package[msg[2]][0][11],self.package[msg[2]][0][12]),
                        analog=[9.91],
                        digital=[0x0001],
                        freq=self.package[msg[2]][1][8]-50,
                        dfreq=5,
                        stat=("ok", True, "timestamp", False, False, False, 0, "<10", 0),
                        soc=int(timestamp),
                        frasec =(int)((20000 * ((self.package[msg[2]][0][4])/20))%1000000)
                        )

            [tag,rtdidMain,rtd_items,rtd_len,seq,ts,interval]=msg[0:7]

            # FIXME: send first item as output
            if rtd_len==4:
                logger.debug("Sending␣to␣output␣0")
                self.send_output(seq,ts,tail[0:rtd_len],0) # 1st output
            else:
                if rtd_len==8:
                    logger.debug("Sending␣to␣output␣1")
                    self.send_output(seq,ts,tail[0:rtd_len],1) # 2nd output

            for j in itemRange:
                sys.stdout.write("%s:%s,%d,%f"%(tag,rtdidMain,xlist[j],tlist[j]))
                for i in range(0,rtd_len):
                     val= tail[j*rtd_len+i]
                     if not math.isnan(val):
                         sys.stdout.write(",%f"%val)
                     else:
                         sys.stdout.write(",")
                sys.stdout.write('\n')

        except Exception as e:
            logging.error("Error␣using␣input␣msg:␣%s"%(str(e)))
            return False
        return self.running

    def quit(self):
        self.dmn.quit()

class rtd_dmn(DMN.DMN):
    'Listening␣to␣a␣multitude␣of␣RTDs'
    def __init__(self, name, bindingRep):
        self.inputStarted=True
        self.last_ts=dict()

        super(rtd_dmn,self).__init__(name,bindingRep)

    def handle_command(self,req):
        '''
        default command handler, should be overriden in subclass
        @req request in json form
        '''
        if req['command']=="startDSP":
            resp="DSP␣started"
        else:
            logging.info("Received␣unknown␣command␣%s"%req)
            resp="unknown␣DSP␣command"
        return resp

    def handle_message(self,tag,msg):
        '''
        default message handler, should be overriden in subclass
        @tag name of this input
        @msg received message from publisher
        '''
        global q

        if not self.inputStarted:
            logging.warning("incoming message when not yet started")
            return

        length=len(msg)
        if length>=4:
            (rtdid,timing,rtd_items,rtd_len)=msg[0:4]
        else:
            logging.warning("not enough fields=%d in message",length)
            return
        rtd_size=rtd_items*rtd_len

        tagid="%s%d"%(tag,rtdid)

        if isinstance(timing,int): # old behavior
            extraFields=length-5
            if timing<=0:
                usec=0
                seq=-timing
            else:
                usec=seq
                seq=None
            if extraFields>0:
                sec=msg[5]
            else:
                sec=None
            if extraFields>1:
                usec=msg[6]
        else: # timing array
            seq=None
            sec=None
            usec=0
            try: # try setting until exception, (u)sec may not be available
                seq=timing[0]
                sec=timing[1]
                usec=timing[2]
            except:
                pass

        if seq!=None:
            interval=1 # default to 1 sequence number interval
            if rtd_items>1:
                seq=seq-rtd_items+1
        if sec!=None:
            ts=sec+usec/1000000 # use timestamp
            default_interval=.0001 # default to low interval until last_ts is available
            interval=default_interval
            if rtd_items>1:
                #logging.debug("rtdid=%d, msg ts=%f,items=%d",rtdid,ts,rtd_items)
                try:
                    last_ts=self.last_ts[tagid]
                    interval=(ts-last_ts)/rtd_items
                    if interval>1: # probably missed a bunch of samples
                        interval=default_interval
                        last_ts=ts-interval*rtd_items
                    else:
                        if interval<0: # time goes backwards
                            logging.warn("rtdid=%s,␣msg␣ts=%f,interval=%f",tagid,ts,interval)
                            interval=default_interval
                    except KeyError: # last_ts is not yet available
                        last_ts=ts-interval*rtd_items
                    self.last_ts[tagid]=ts
                    ts=last_ts+interval
        else:
            ts=None

        value=msg[4]
        if isinstance(value,list):
            q.put([tag,rtdid,rtd_items,rtd_len,seq,ts,interval]+value)
        else:
            q.put([tag,rtdid,rtd_items,rtd_len,seq,ts,interval,value])

    def sendOutput(self, rtdid, seq, ts, values,outputIndex=0):
        '''
        send values as DSP output (non-aggregated, rtd_items=1)
        @param seq the seqence number for the result, or None when sequence is not known
        @param ts  the timestamp for the result
        @param values an array of values, or single value containing the result
        @param outputIndex the index of the output
        '''
        sec=int(ts)
        usec=int((ts-sec)*1000000)
        msg=[rtdid,[seq,sec,usec],1,len(values),values]
        self.send_output(msg, outputIndex)

    def quit(self):
        self.msgr.send_command(self.bindingReq, '{␣"command":"stop"␣}')

def dsp_realtime(dsp_name):
    #dsp=RTD_DSP(dsp_name,"ipc:///tmp/dsp%d"%(randint(0,100)))
    dsp=RTD_DSP(dsp_name)
    variant="dsp"
    # read configuration
    configfile=path.join(path.dirname(path.realpath(__file__)),"rtd.properties")
    defaults={ "sensors":["sensor1"],"outputs":[] }
    config=ConfigParser(defaults)
    config.read(configfile)
    dsp.dmn=rtd_dmn(dsp_name,"inproc://tmp/%s"%(dsp_name))

    sensors=json.loads(config.get("defaults","sensors"))
    # listen to realtime data
    for sensor in sensors:
        ports=json.loads(config.get(sensor,variant+"_ports"))
        host=config.get(sensor,"tcp_host")
        try:
            tag=config.get(sensor,"tag")
        except Exception as e:
            logging.error("Error␣config␣msg:␣%s"%(str(e)))
            tag=host
        for port in ports:
            dsp.dmn.add_subscription(tag,"tcp://%s:%d"%(host,port))

```

```python
    outputs=json.loads(config.get("defaults","outputs"))
    outputIndex=0
    for output in outputs:
        binding=config.get(output,variant+"_binding")
        rtdid=config.get(output,variant+"_rtdid")
        if isinstance(rtdid,str):
            rtdid=int(rtdid)
        logging.info("Adding␣binding␣%s␣to␣output␣with␣rtdid␣%d"%(binding,rtdid))
        dsp.add_output(rtdid,binding,outputIndex)
        outputIndex+=1

    return dsp

def dsp_all():
    global q,running,stopping
    stopping=False
    signal.signal(signal.SIGINT, signal_handler)
    q = queue.Queue()
    dsp=dsp_realtime('MyDSP')
    running=True


    pmu_id=1
    data_rate=50
    pmu = Pmu(ip="131.180.164.101", port=4702, pmu_id=1, data_rate=50, set_timestamp=False)
    #pmu = Pmu(ip="131.180.164.99", port=4701, pmu_id=1, data_rate=50, set_timestamp=False)
    pmu.logger.setLevel("INFO")

    cfg = ConfigFrame2(pmu_id,  # PMU_ID
                       1000000,  # TIME_BASE
                       1,  # Number of PMUs included in data frame
                       "sensorPMU",  # Station name
                       pmu_id,  # Data-stream ID(s)
                       (False, True, True, True),
                       3,  # Number of phasors
                       1,  # Number of analog values
                       1,  # Number of digital status words
                        ["VA", "VB", "VC", "ANALOG1", "BREAKER␣1␣STATUS",
                        "BREAKER␣2␣STATUS", "BREAKER␣3␣STATUS",
                        "BREAKER␣4␣STATUS", "BREAKER␣5␣STATUS",
                        "BREAKER␣6␣STATUS", "BREAKER␣7␣STATUS",
                        "BREAKER␣8␣STATUS", "BREAKER␣9␣STATUS",
                        "BREAKER␣A␣STATUS", "BREAKER␣B␣STATUS",
                        "BREAKER␣C␣STATUS", "BREAKER␣D␣STATUS",
                        "BREAKER␣E␣STATUS", "BREAKER␣F␣STATUS",
                        "BREAKER␣G␣STATUS"],  # Channel Names
                       [(1, "v"), (1, "v"), (1, "v")],
                       [(1, "pow")],  # Conversion factor for analog channels
                       [(0x0000, 0xffff)],  # Mask words for digital status words
                       data_rate,  # Nominal frequency
                       1,  # Configuration change count
                       50)  # Rate of phasor data transmission)


    pmu.set_configuration(cfg)
    pmu.set_header("sensorPMU␣here!")

    pmu.run()
    print("pmu␣started")
    while (running):
        running=dsp.updateInput(q,pmu)
        if stopping:
            logger.debug('Trying␣to␣stop␣after␣CTRL-C!')
            running=False


    logger.debug('Trying␣to␣quit␣after␣CTRL-C!')
    dsp.quit()
    logger.debug('Quited␣after␣CTRL-C!')

def signal_handler(signal, frame):
    global stopping
    logger.info('You␣pressed␣Ctrl+C!')
    stopping=True

if __name__ == "__main__":
    logfile='session-dsp.log'
    logger = logging.getLogger('rtd_dsp')
    logging.basicConfig(level=logging.INFO,
                        format='%(asctime)s.%(msecs)03d␣%(levelname)-8s␣%(message)s',
                        datefmt='%Y-%m-%d,%H:%M:%S',
                        filename=logfile,
                        filemode='w')
    stderrLogger=logging.StreamHandler()
    stderrLogger.setLevel(logging.INFO)
    stderrLogger.setFormatter(logging.Formatter(logging.BASIC_FORMAT))
    logger.addHandler(stderrLogger)

    dsp_all()

    logging.info("Stopped,␣running=%d"%(running))
```

Bibliography

[1] V. Terzija et al. “Wide-Area Monitoring, Protection, and Control of Future Electric Power Networks”. In: *Proceedings of the IEEE* 99.1 (Jan. 2011), pp. 80–93. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2060450.

[2] *EPFL Smart Grid Project EPFLSmartGrid*. URL: https://smartgrid.epfl.ch/.

[3] *About The Project - OpenPMU*. URL: https://sites.google.com/site/openpmu/project-definition.

[4] Paolo Romano and Mario Paolone. “DFT-based Synchrophasor Estimation Algorithms and their Integration in Advanced Phasor Measurement Units for the Real-time Monitoring of Active Distribution Networks”. PhD thesis.

[5] A. G. Phadke and J. S. Thorp. *Synchronized phasor measurements and their applications*. SPRINGER INTERNATIONAL PU, 2008.

[6] “IEEE Standard for Synchrophasor Measurements for Power Systems”. In: *IEEE Std C37.118.1-2011 (Revision of IEEE Std C37.118-2005)* (Dec. 2011), pp. 1–61. DOI: 10.1109/IEEESTD.2011.6111219.

[7] Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signals and systems*. Prentice-Hall, 1983.

[8] P. Romano and M. Paolone. “Enhanced Interpolated-DFT for Synchrophasor Estimation in FPGAs: Theory, Implementation, and Validation of a PMU Prototype”. In: *IEEE Transactions on Instrumentation and Measurement* 63.12 (Dec. 2014), pp. 2824–2836. ISSN: 0018-9456. DOI: 10.1109/TIM.2014.2321463.

[9] D. Hart et al. “A new frequency tracking and phasor estimation algorithm for generator protection”. In: *IEEE Transactions on Power Delivery* 12.3 (July 1997), pp. 1064–1073. ISSN: 0885-8977. DOI: 10.1109/61.636849.

[10] I. Carugati et al. “Three-phase harmonics measurement method based on mSDFT”. In: *IEEE Latin America Transactions* 12.7 (Oct. 2014), pp. 1250–1257. ISSN: 1548-0992. DOI: 10.1109/TLA.2014.6948860.

[11] Jun-Zhe Yang and Chih-Wen Liu. “A precise calculation of power system frequency”. In: *IEEE Transactions on Power Delivery* 16.3 (July 2001), pp. 361–366. ISSN: 0885-8977. DOI: 10.1109/61.924811.

[12] Dinesh Rangana Gurusinghe, Dean Ouellette, and Athula D Rajapakse. “Implementation of Smart DFT-based PMU Model in the Real-Time Digital Simulator”. In: (2011).

[13] P. Castello et al. “Chapter 5 - Hardware for PMU and PMU Integration”. In: *Phasor Measurement Units and Wide Area Monitoring Systems*. Ed. by Antonello Monti, Carlo Muscas, and Ferdinanda Ponci. Academic Press, 2016, pp. 63–86. ISBN: 978-0-12-804569-5. DOI: https://doi.org/10.1016/B978-0-12-804569-5.00005-7. URL: http://www.sciencedirect.com/science/article/pii/B9780128045695000057.

[14] L. Peretto and R. Tinarelli. *Sensors for PMUs*. Elsevier Inc., 2016, pp. 53–62. ISBN: 9780128045695. DOI: 10.1016/B978-0-12-804569-5.00004-5. URL: http://dx.doi.org/10.1016/B978-0-12-804569-5.00004-5.

[15] “Optical Atomic Standards”. In: *Time - From Earth Rotation to Atomic Physics*. Wiley-Blackwell, 2010. Chap. 11, pp. 181–187. ISBN: 9783527627943. DOI: 10.1002/9783527627943.ch11. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9783527627943.ch11. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9783527627943.ch11.

[16] Power System et al. *IEEE Standard for Synchrophasor Data Transfer for Power Systems*. Vol. 2011. December. 2011, pp. 1–53. ISBN: 9780738168135. DOI: 10.1109/IEEESTD.2011.6111222.

[17] Pieter Hintjens. *ZeroMQ Messaging for Many Applications*. OReilly Associates, 2012.

[18] *CBOR*. URL: http://cbor.io/.

[19] *Build IBCrazy's Cloverleaf - The ultimate circularly polarized aerial antenna!* URL: https://www.rcgroups.com/forums/showthread.php?1388264-Build-IBCrazy-s-Cloverleaf-Theultimate-circularly-polarized-aerial-antenna!.

[20] S. Šandi, B. Krstajić, and T. Popović. "pyPMU x2014; Open source python package for synchrophasor data transfer". In: *2016 24th Telecommunications Forum (TELFOR)*. Nov. 2016, pp. 1–4. DOI: 10.1109/TELFOR.2016.7818916.

[21] GridProtectionAlliance. *GridProtectionAlliance/PMUConnectionTester*. May 2018. URL: https://github.com/GridProtectionAlliance/PMUConnectionTester.