A Computer Algorithm

for

Implementing

A Frequency Relay

by

© Donny H.W. Kwan

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirement for the degree of

Master of Science

in Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

Electrical and Computer
Engineering

Winnipeg, Manitoba, Canada
April 1991.

ISBN 0-315-76958-0

A COMPUTER ALGORITHM FOR IMPLEMENTING A FREQUENCY RELAY

BY

DONNY H.W. KWAN

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

The objective of this thesis is to develop a computer algorithm for implementing a frequency relay. The algorithm aims to track the frequency of power system more accurately and faster than a conventional method. The modified Discrete Fourier Transform technique has been tested with typical power system conditions on a Unix workstation, and it works very well to signal undergoing frequency swing, corrupted with harmonics and random noise, shifting phase, as well as sudden dropping its voltage magnitude. However, the input signal for the new method has to be anti–aliased and limited to 10 Hz/sec for its rate of change of frequency.

Acknowledgements

I would like to express my sincere thanks to Professor Glenn Swift for his guidance, wisdom and support. It is my honor to have such a fine advisor. I would also like to thank Professor Peter McLaren, Mr. Dave Fedirchuk, Tom Gouldsborough and Doug Chapman for their helpful suggestions to this thesis. I wish to thank my parents for their patience and understanding. Finally, I would like to thank Manitoba Hydro for its financial support to this research work.

Table of Content

Abstract i

Acknowledgements ii

List of Figures v

Chapter 1. Introduction 1
1.1. Objective 1
1.2. Problems 1
1.3. Scope 2

Chapter 2. Background 3
2.1. A Power System Protection Scheme 3
2.2. Development of Digital Relaying 4
2.3. Mathematical Basis of The Fourier Series 6
2.4. Digitized Sample Signal 8
2.5. Nyquist Criterion 10
2.6. Recursive Phasor Computation 10
2.7. The Relationship Between the Frequency Deviation (Δf) and the Rate of Change of Phase Angle 13
2.8. FFT Versus DFT 15

Chapter 3. The New Algorithm 17
3.1. Using Half of the Window Data 17
3.2. Three Phase Voltage Input Signals 18
3.3. Rate of Change of Frequency Limiter 19
3.4. Computer Software 19
3.5. Program Flow-Chart 19

Chapter 4. Typical Test Signals 23
4.1. Frequency Swing Signal 23
4.1.1. How to Define a Signal with Time-varying Frequency 24
4.2. Frequency Swing Signals with 10% 3rd Harmonic (in magnitude) at Various Overlapping Angles. 28
4.3. Frequency Swing Signal with 10% 3rd and 1% 5th Harmonics in Phase with the Fundamental 30
4.4. Frequency Swing Signal with Gaussian Distributed High Frequency Random Noise 31
4.4.1. Gaussian Distributed Random Noises 32
4.4.2. Low Pass Digital Filters 34
4.5. Fundamental Voltage Signal (60 Hz) with 50% Sudden Drop in Magnitude 37
4.5.1. Voltage Signal Before and During the Fault 37

4.6. Fundamental Signal That Change Phase by -90 Degrees Over A Period of One Cycle 39

Chapter 5. Test Results 41

5.1. Frequency Swing Signal 41

5.1.1. Three-Phase Versus One-Phase 41

5.2. Frequency Swing Signal with 10% 3rd Harmonic (in magnitude) at Various Overlapping Angles. 43

5.3. Frequency Swing Signal with 10% 3rd and 1% 5th Harmonics. 49

5.4. Frequency Swing Signal with Gaussian Distributed High Frequency Random Noise 51

5.5. Fundamental Voltage Signal (60 Hz) with 50% Sudden Drop in Magnitude 54

5.6. Fundamental Signal That Change Phase by -90 Degrees Over A Period of One Cycle 56

Chapter 6. Considerations on the Hardware and Software Implementation 58

6.1. Microprocessor Board Implementation Consideration 58

6.2. Testing the Algorithm with A Larger Frequency Swing Range. 60

6.3. Adaptive Relaying 61

Chapter 7. Conclusions 63

References 64

Appendices 66

List of Figures

Figure 1. A Power Protection System 3
Figure 2. A Digital Relay Configuration 5
Figure 3. Digitized Samples 8
Figure 4. Sampling Window and Reference Phasors 11
Figure 5. Relationship between the Change of Frequency and the Phasor Angle 13
Figure 6. Change in Phasor Angle when the Frequency Deviates from 60 Hz 15
Figure 7. Flow Chart in Description 21
Figure 8. Flow Chart in Equations 22
Figure 9. Frequency Swing 27
Figure 10. Rate of Change of Frequency 27
Figure 11. Three-phase Voltage Signals 28
Figure 12. Voltage Signals with 10% 3rd Harmonics at $\gamma = 90^0$ 30
Figure 13. Voltage Signals with 10% 3rd and 1% 5th Harmonics at $\gamma = 0^0$ ... 31
Figure 14. Configuration of Computer Program Simulation 32
Figure 15. Gaussian Distributed Probability Graph 33
Figure 16. Computer Generated Gaussian Distributed Random Noise Waveform 33
Figure 17. Block Diagram of a Low-Pass (Integral) Filter. 35
Figure 18. Characteristics of a Low-Pass Integral Filter 36
Figure 19. Power System at Fault 37
Figure 20. Fundamental Voltage Signals with 50% Drop in Magnitude 38
Figure 21. Phase Angle Characteristics of a Voltage Signal 39
Figure 22. Fundamental Voltage Signal that Shift -90 degrees over one cycle 40
Figure 23. Result: Frequency Response of Pure Power Swing Three-Phase Signals 42
Figure 24. Results: One-phase Response 42
Figure 25. Result: Rate of Change of Frequency 43
Figure 26. Voltage Signals with 10% 3rd Harmonic at $\gamma = 0^0$ 44
Figure 27. Result: 10% 3rd Harmonic at $\gamma = 0^0$ 44
Figure 28. Voltage Signals with 10% 3rd Harmonic at $\gamma = 15^0$ 45
Figure 29. Result: 10% 3rd Harmonic at $\gamma = 15^0$ 45
Figure 30. Voltage Signals with 10% 3rd Harmonic at $\gamma = 45^0$ 46
Figure 31. Result: 10% 3rd Harmonic at $\gamma = 45^0$ 46
Figure 32. Voltage Signals with 10% 3rd Harmonic at $\gamma = 60^0$ 47
Figure 33. Result: 10% 3rd Harmonic at $\gamma = 60^0$ 47
Figure 34. Voltage Signals with 10% 3rd Harmonic at $\gamma = 90^0$ 48
Figure 35. Result: 10% 3rd Harmonic at $\gamma = 90^0$ 48
Figure 36. Voltage Signals with 10% 3rd and 1% 5th Harmonics at $\gamma = 0^0$ .. 50
Figure 37. Result: 10% 3rd and 1% 5th Harmonics at $\gamma = 0^0$ 50
Figure 38. Voltage Signals with Gaussian Distributed High Frequency Random Noises 52
Figure 39. Result: Without Digital Low-Pass Filter: Gaussian Distributed High Frequency Random Noises 52

Figure 40. Result: With Digital Low-Pass Filter: Gaussian Distributed High Frequency Random Noises 53
Figure 41. Result: Averaging the Output + Digital Low Pass Filter 53
Figure 42. Fundamental Voltage Signals with 50% Sudden Drop in Magnitude at Time = 0 seconds. 55
Figure 43. Result: 50% Sudden Drop in Magnitude 55
Figure 44. Fundamental Voltage Signals That Change Phase by -90 Degrees Over A Period of One Cycle. 57
Figure 45. Result: Fundamental Signals That Change Phase by -90 Degrees 57
Figure 46. Modified Algorithm for Microprocessor Implementation 59
Figure 47. Frequency Response with Range of Swing of 20 Hz. 60
Figure 48. Frequency Response with Range of Swing of 20 Hz (Scale Enlarged) 61
Figure 49. Frequency Response Feedback Sampling Rate. 62
Figure 50. Frequency Response Feedback Sampling Rate (Scale Enlarged). . 62

Chapter 1. Introduction

1.1. Objective

With the increasing complexities of modern power systems, a more accurate and faster protection scheme is needed. It is thus the objective of this thesis to present a computer relaying algorithm to implement a frequency relay, which will accomplish the above goal.

1.2. Problems

Conventional electro–mechanical frequency relays are still in operation on many power systems. However, due to the rapid development in computing processor technology, digital frequency relays are gradually replacing the conventional ones.

Many algorithms have been used to detect the system frequency, among these, zero–crossing and Discrete Fourier Transform (DFT) are the two most popular techniques which engineers prefer. The zero–crossing method obtains the frequency by calculating the inverse of the measured period. However multiple–crossings, caused by the presence of noise and harmonics, affect its accuracy. Filtering alone cannot solve the problem without significant time delay in the response.

Although many ideas have been published based on the DFT algorithm to obtain the system frequency by measuring the voltage or current phasors, no single one seems to be satisfactory. They either fail to substantially reduce the delay time, or do not test the algorithm thoroughly with typical power system operating conditions. Consequently, a rather simple algorithm will be developed for a frequency relay. The algorithm is to be tested under typical system operating conditions as suggested by Manitoba Hydro.

1.3. Scope

In this thesis, we are going to investigate a computer algorithm for implementing a frequency relay which will detect the system frequency. We first discuss the background of the development of digital relaying. Secondly, the theory of the DFT algorithm and how the system voltage inputs are sampled is explained. Thirdly, we will then deal with all of the typical test signals and methods to create time–varying frequency signals. Fourthly, test results and considerations on hardware and software implementation will be discussed. The final chapter will be the conclusions.

Chapter 2. Background

2.1. A Power System Protection Scheme

A power protection system [1] [2] basically consists of three subsystems:

1. Transducers (T)
2. Relays (R)
3. Circuit Breakers (CB)

Transducers provide the inputs in the form of voltages and current signals to the relays. Relays are the devices which sense a fault and energize the circuit breakers to open the contacts in order to isolate the fault. Circuit breakers disconnect the appropriate lines during the disturbances in order to prevent total collapse of the system. Fast and accurate elimination of a fault requires correct operation of all of the three subsystems.

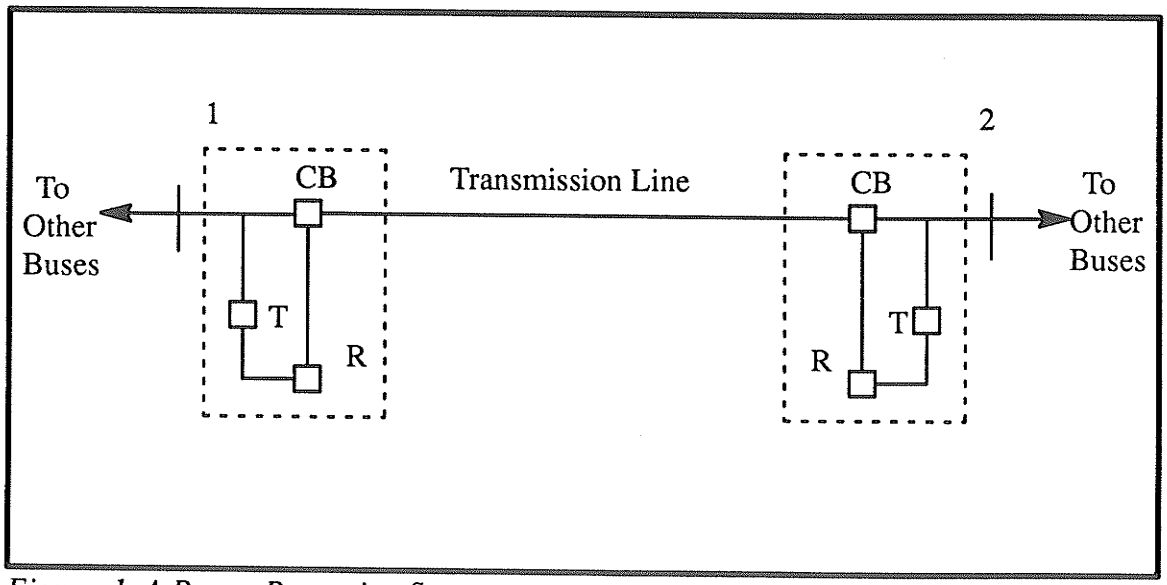


Figure 1. A Power Protection System

Figure (1.) shows a power protection system for the transmission line 1–2. Two identical protection systems, enclosed by dotted lines, are located at both ends of the transmission line.

According to Warrington's summary [3] [4], the general requirements of a relay are:

1. The relay should operate and respond to the type of fault which it is intended to protect against, and not for any other conditions.
2. The relay must be immune from transient effects, such as, a drop in voltage, peak currents, direct current components, harmonic content and noise.
3. The relay should have a range of adjustment to permit it to operate selectively with other relays.
4. The construction of the relay must be simple and accessible, so as to facilitate testing and regular maintenance work.
5. The construction should allow easy modifications to meet unusual conditions of temperature, humidity, vibration, mechanical shock, etc.

2.2. Development of Digital Relaying

The conventional electromagnetic relays have been used in the power system schemes for decades. Despite the fact that the idea of the digital relay was introduced by some experts in late 1960's, it was not until several years ago that the implementation of digital relays became cost effective as well as technologically feasible. Since the 1960's, the cost of hardware processors has been substantially reduced and many algorithms have been proposed. The combination of rapid development in both computer hardware and software makes it now possible to manufacture digital relays commercially. Computer relaying is going to contribute to all aspects of real time monitoring and control of power systems.

Figure (2.) shows a simplified digital relay configuration [5]. Analog three–phase voltage and current signals are acquired through transducers such as current transformers or

capacitor voltage transformers. The input signals are filtered with anti–aliasing filters. Analog–to–digital converters are then used to sample those quantities simultaneously, converting the analog signals to digital signals. The digital computer imports the samples and makes decisions according to the algorithm. The results are sent out from the processor to the control center and breaker are tripped if necessary.

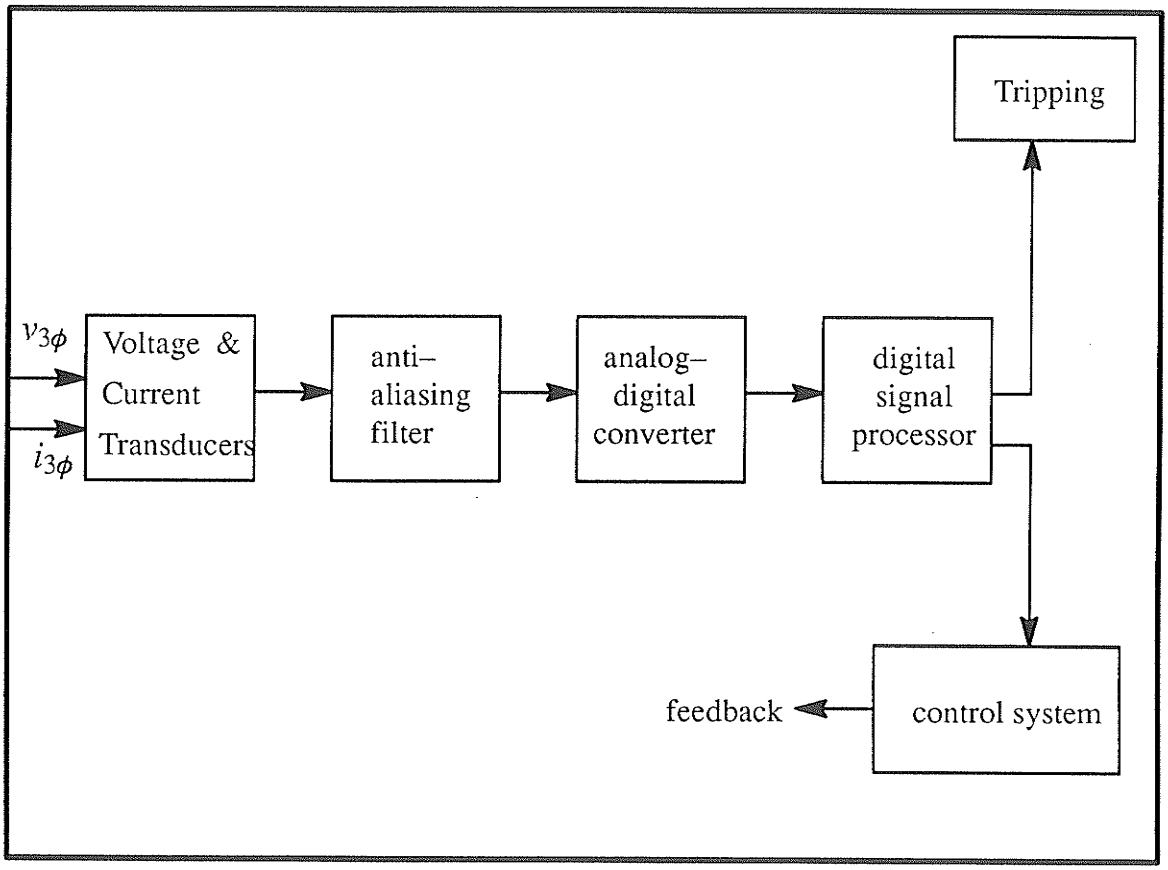


Figure 2. A Digital Relay Configuration

The microprocessor based devices, which are usually installed in substations, are constantly processing data received from the power system. It has functions of monitoring and control. The digital relay is considered to be a measuring system which computes the input data and makes a decision according to pre–set criteria. Digital devices are also more reliable because of the digital equipment's self–diagnostic ability.

A conventional relay is usually designed to detect a specific fault in the power system, and cannot dynamically switch to another type of relay. With a digital computer, many types of relay algorithms are stored in the processor's memory and hence, multi–function parallel processing of adaptive algorithms on the digitized signals is possible. However, this thesis will only deal with one algorithm for the frequency relay.

The normal operating frequency of a typical power system is 60 Hz (50 Hz in some countries) and it deviates when there is a generation–load unbalance. Overfrequency occurs if the system is underloaded and underfrequency if overloaded. Frequency relays detect any frequency deviation and, if necessary, disconnect load blocks in order to restore the system to a balanced condition and hence prevent the total collapse of the system.

A frequency relay is particularly more useful than an impedance relay during a power swing.This is because the impedance seen by a relay may fall to a low value during the power swing, even if no fault occurs, causing false tripping.

With an appropriate algorithm and design, a digital frequency relay has the following advantages over a conventional relay: greater economy, better performance, and greater reliability and flexibility.

Accuracy and speed are the two most important criteria for a relay algorithm. Since the frequency information is to be used in a feedback control loop in the power system, the delay time has to be minimized. Accuracy, however, cannot be compromised for security.

2.3. Mathematical Basis of The Fourier Series

The Fourier Series is one of the most commonly used algorithms in computer relaying, because it provides a technique for extracting fundamental and harmonic frequencies of incoming signals. The nature of these fundamental and non–fundamental frequency signals has an important bearing on the performance of relaying algorithms.

The input signals of relays encountered in power systems, such as phase voltages, are essentially periodic and the ideal frequency of the system in steady state is 60 Hz [6] [7].

A voltage signal $\vec{v}(t)$ is periodic if there exits a τ such that

$$\vec{v}(t) = \vec{v}(t+\tau) \quad ; \quad \text{for } all \ t \tag{1.}$$

The fundamental period τ_o of the voltage signal is the smallest positive value of τ such that (1.) holds. The fundamental frequency ω_o is defined by

$$\omega_o = \frac{2\pi}{\tau_o} \tag{2.}$$

Any finite sum of the form

$$\vec{v}(t) = C_o + C_1 \ e^{j\omega_o t} + C_2 \ e^{j2\omega_o t} + \ \ldots\ldots + C_N \ e^{j(N-1)\omega_o t} \tag{3.}$$

$$\vec{v}(t) = \sum_{k=0}^{k=N-1} C_k \ e^{jk\omega_o t} \tag{4.}$$

where $C_0, C_1, C_2, \ldots C_N$ are constant coefficients

will be periodic with fundamental frequency ω_o. A Fourier analysis aims to decompose an arbitrary periodic signal into spectral components as in equation (4.). The Discrete–Time Fourier Transform is represented by the following equation.

$$\vec{v}(\omega) = \sum_{i=0}^{N-1} v(iT) \ e^{-ji\omega T} \tag{5.}$$

where N is the number of samples taken over one period

T is the sampling period

i is the integer number (NOT a complex constant here)

j is the complex constant of $\sqrt{-1}$

2.4. Digitized Sample Signal

If we take N samples of a continuous signal $v(t)$ at interval of T seconds to form a finite duration discrete–time signal as shown in Figure (3.)

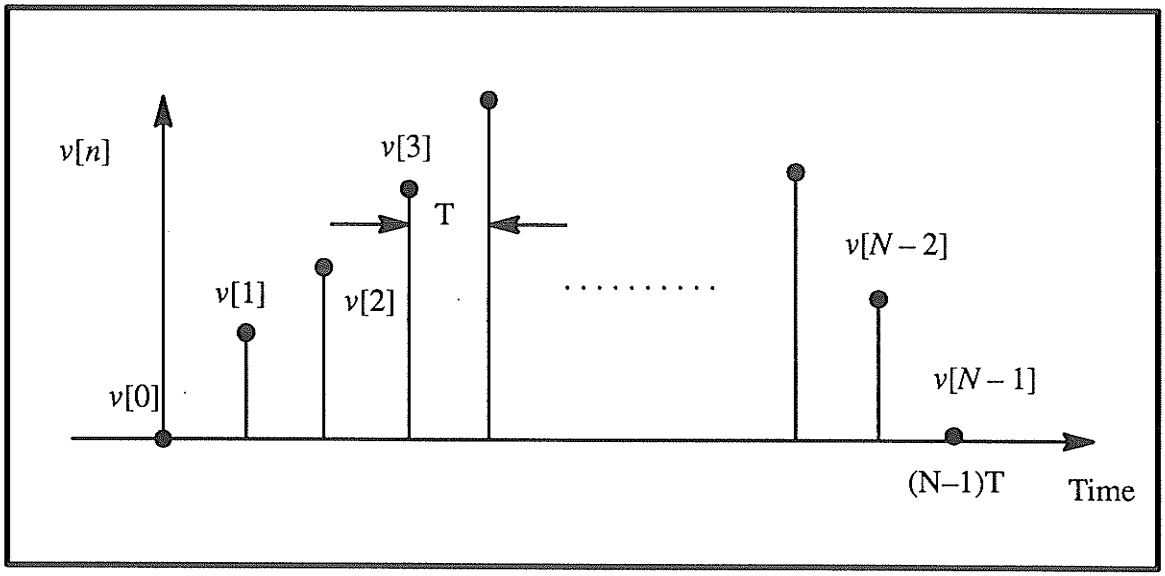


Figure 3. Digitized Samples

with $$\vec{v}[n] = \vec{v}(nT) \; ; \quad \text{for } 0 < n < N-1 \tag{6.}$$

The finite description of the transform is a sampled version of the Discrete Fourier Transform (DFT):

$$\vec{v}(h\Omega_{o)} = \sum_{i=0}^{N-1} v(iT) \; e^{-jhi\Omega_o T} \tag{7.}$$

where Ω_o is the fundamental frequency of discrete samples.

h is the harmonic order.

We are interested here in the simplest form of the Fourier algorithm, which extracts the fundamental frequency phasor from samples of a periodic signal over a period or less than a period. The discrete correlation of samples of a reference sinusoid and cosinusoid with the input signals is called Discrete Fourier Transform.

Consider an ideal sinusoidal voltage signal of frequency f

$$\vec{v}(t) = \sin(2\pi ft + \phi) \tag{8.}$$

where f is the frequency in Hz

ϕ is an arbitrary phase angle.

The signal is sampled at a rate of N times per cycle of 60 Hz (f) waveform. The period of the sampling rate is

$$\frac{1}{fN}$$

The sample set { $\vec{v}^k$ } at window k would be

$$\vec{v}^k = \sum_{i=k-N+1}^{k} \sin(2\pi f \frac{1}{fN} i + \phi)$$

$$\vec{v}^k = \sum_{i=k-N+1}^{k} \sin(\frac{2\pi}{N} i + \phi) \tag{9.}$$

The DFT of { $\vec{v}^k$ } containing a fundamental frequency component, at window k is;

$$\vec{v_1}^k = \sum_{i=k-N+1}^{k} v_i \; e^{-j\frac{2\pi}{N}i} \quad ; \; v_i \text{ is the sampled version of } \vec{v}(t)$$

$$\vec{v_1}^k = \sum_{i=k-N+1}^{k} v_i \; \sin\frac{2\pi}{N} i + j \sum_{i=k-N+1}^{k} v_i \; \cos\frac{2\pi}{N} i \tag{10.}$$

where subscript 1 indicates the fundamental component.

That is, equation (10.) identifies the fundamental frequency component of the sample set $\{ \vec{v}^* \}$, which might contain some frequency components other than fundamental under the real situation. One other important consideration is that the input signal has to be band–limited to prevent aliasing, which causes error in the DFT algorithm, due to the Nyquist Criterion.

2.5. Nyquist Criterion

The Nyquist criterion states that the minimum sampling rate, $f_{sampling}$, needed to reconstruct a band–limited waveform, f_{signal} , without error is given by 2 x f_{signal} [7].

If $f_{sampling} < 2B$, where B is the highest significant frequency component in the sampled waveform, an aliasing error will occur. Aliasing is the distortion or interference caused by the overlap between the lower lobe of the spectrum centered on $f_{sampling}$ and the baseband spectrum. The aliasing error can be reduced by either using a higher sampling frequency or a presampling low–pass filter. It should also be noted that the highest frequency component that can be evaluated with an N–point DFT is $f = \frac{f_{sampling}}{2} = \frac{Nf_0}{2}$, where f_0 is the fundamental frequency.

2.6. Recursive Phasor Computation

As mentioned earlier, the output from the relay my be used for a control feedback loop. Therefore, reducing the time delay is one of the important criteria in our algorithm. The Recursive phasor calculation reduces computation time extensively. We use N=4 samples per cycle to simplify the illustration of recursive phasor computation.

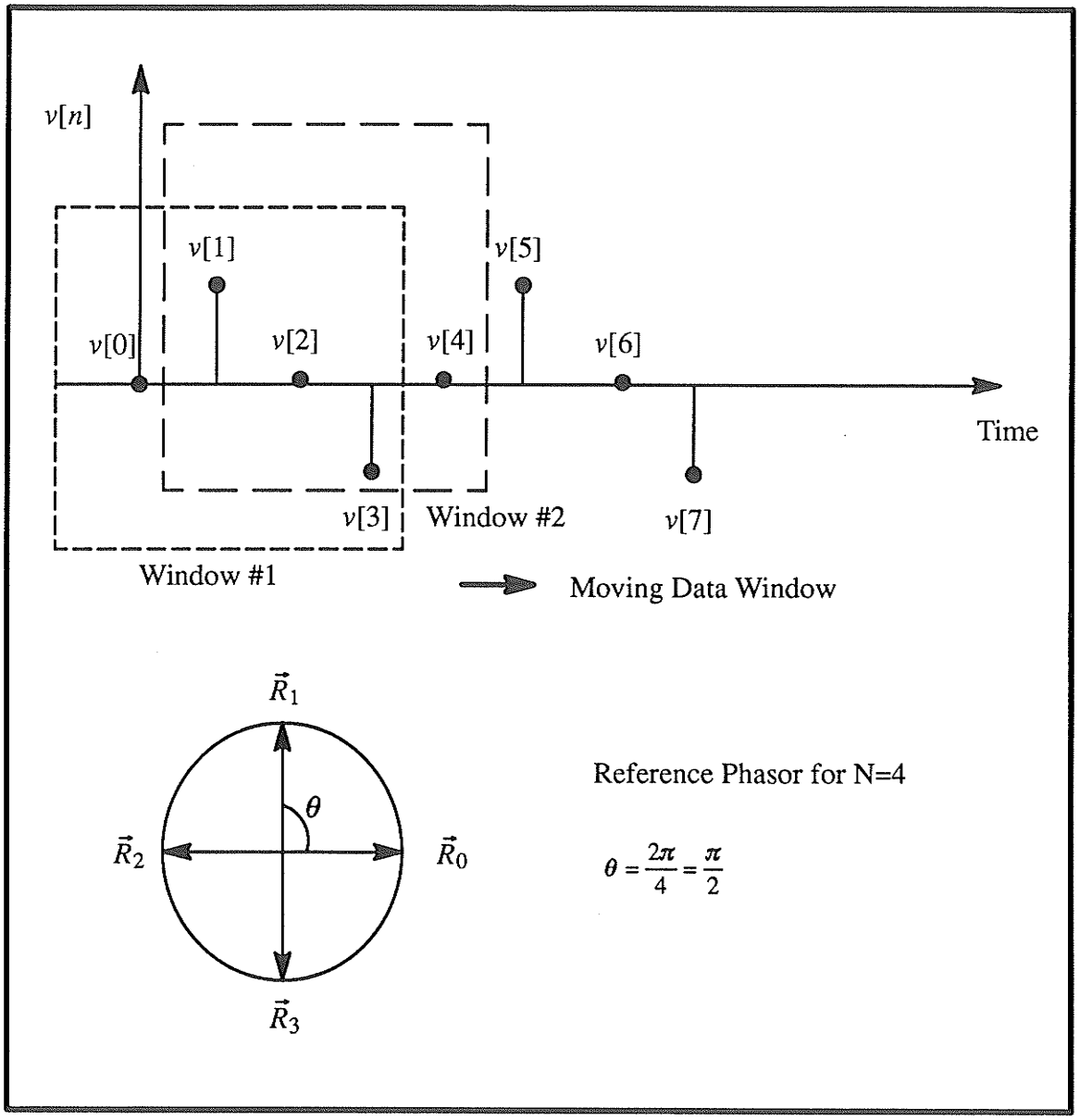


Figure 4. Sampling Window and Reference Phasors

Referring to Figure (4.), $v[0]$, $v[1]$, $v[2]$, $v[3]$, are the data set obtained at window #1, $v[1]$, $v[2]$, $v[3]$, $v[4]$, at window #2, etc. $\vec{R}_0$, $\vec{R}_1$, $\vec{R}_2$, $\vec{R}_3$, are the four reference vectors for the DFT correlation. Using equation (10.), the fundamental component of the incoming signal at window k is

$$\vec{v_1}^k = \sum_{i=k-N+1}^{k} v_i \; e^{-j\frac{2\pi}{N}i}$$

for N=4;

$$\vec{v_1}^k = \sum_{i=k-3}^{k} v_i \; e^{-j\frac{\pi}{2}i} \tag{11.}$$

for window k=1 and k=2, the DFT correlation vector are:

$$\vec{v}_1{}^{\#1} = v[0]\vec{R}_0 + v[1]\vec{R}_1 + v[2]\vec{R}_2 + v[3]\vec{R}_3 \tag{12.}$$

$$\vec{v}_1{}^{\#2} = v[1]\vec{R}_1 + v[2]\vec{R}_2 + v[3]\vec{R}_3 + v[4]\vec{R}_0 \tag{13.}$$

It is noted that three out of four terms are common to equations (12.) and (13.) As the window is progressing from window #1 to #2, only one sample, $v[0]$, is discarded and one sample, $v[3]$, is added to the new data set. By retaining three common terms in memory and adding the new term for the next phasor calculation, computation time is greatly reduced.

For a pure sinusoidal input signal of fundamental frequency,

$$v[N+r] = v[r]; \quad \text{for } all \;\; r,$$

and equation (11.) becomes;

$$\vec{v}_1{}^{\,r} = \vec{v}_1{}^{\,(r-1)} \tag{14.}$$

Equation (14.) [8] indicates that a stationary phasor in the complex plane is obtained when the recursive computation is applied to the pure sine waveform of the fundamental frequency signal.

2.7. The Relationship Between the Frequency Deviation (Δf) and the Rate of Change of Phase Angle

The frequency deviation of a voltage signal can be determined from the rate of change of phase angle of the voltage phasor using the following equations.

$$\frac{d\psi}{dt} = 2\pi \ \Delta f \tag{15.}$$

or $\frac{\Delta\psi}{\Delta t} = 2\pi \ \Delta f$ for a finite time interval Δt between samples.

Thus, $\Delta f = \frac{1}{2\pi}\frac{\Delta\psi}{\Delta t} = \frac{1}{2\pi}\frac{\psi_r - \psi_{r-1}}{\left(\frac{1}{60N}\right)}$ (16.)

where $\Delta\psi$ is the change of phase angle between successive measurement

Δt is the change of time

Δf is the frequency deviation from 60 Hz

N is the number of samples taken every period.

ψ_r and ψ_{r-1} are consecutive calculated values of ψ.

The following example illustrates the above concept, with N=1, that is, just one phase measurement per cycle;

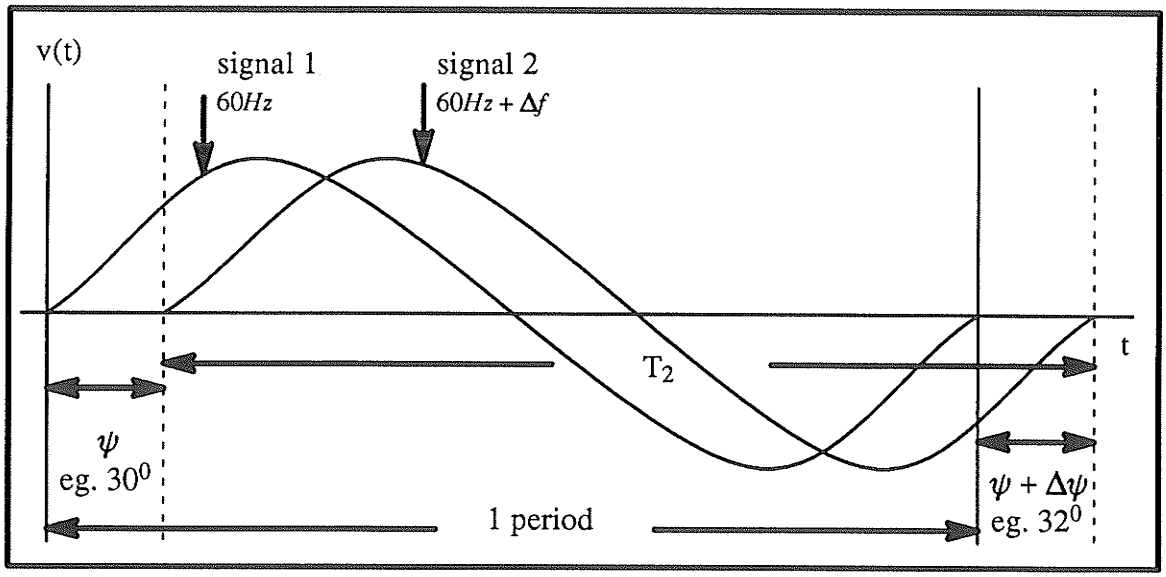


Figure 5. Relationship between the Change of Frequency and the Phasor Angle

Signal 1 in Figure (5.) is the reference phasor with frequency of 60 Hz, while signal 2 changes its frequency from 60 to 60+Δf at the end of the period. By applying equation (15.), we get,

$$\frac{d\psi}{dt} = \frac{\Delta\psi}{\Delta t} = \frac{30^0 - 32^0}{1/60} = -120 \ \ deg/sec = -\frac{2\pi}{3} rad/sec \ \ or \ \ \Delta f = -\frac{1}{3} Hz = -0.33Hz$$

Hence, the frequency of signal 2 is 60–0.33 =59.67 Hz. Or, we can just obtain the frequency of the signal 2 by taking the inverse of the period T_2.

$$T_2 = \frac{362}{360} x \frac{1}{60} \qquad f_2 = \frac{1}{T_2} = 59.67Hz$$

The rate of change of the phasor angle is thus directly proportional to the input signal frequency deviation. Figure (6.) shows the phasor rotates away from the reference phasor when the signal frequency deviates from the fundamental. The frequency of a signal can thus be calculated using equation (16.) and the equation $f_{signal} = 60 + \Delta f$.

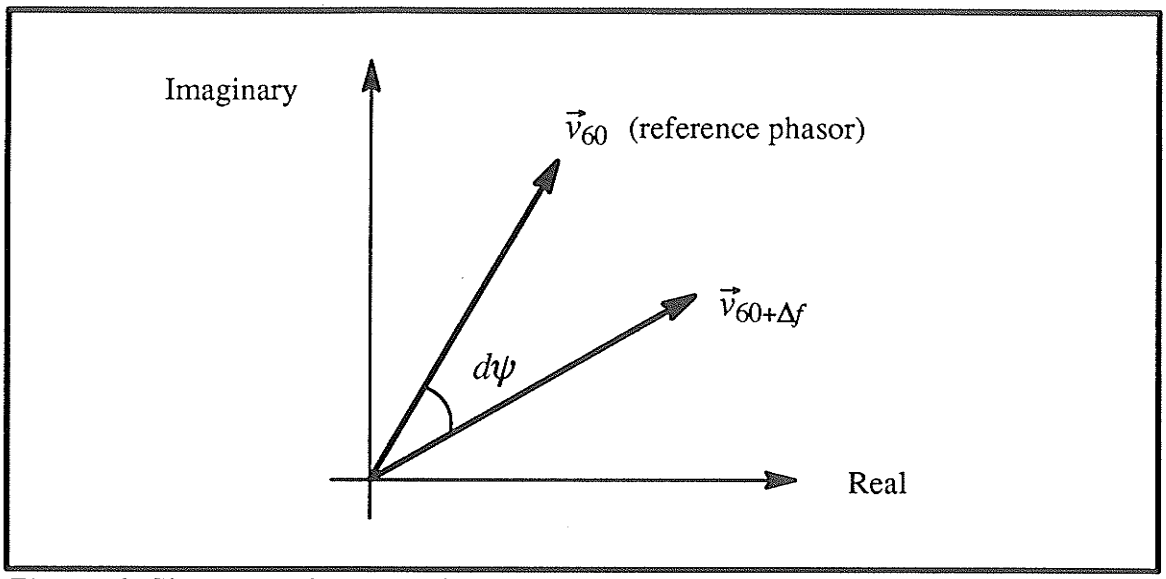


Figure 6. Change in Phasor Angle when the Frequency Deviates from 60 Hz

2.8. FFT Versus DFT

The Fast Fourier Transform (FFT) is a numerical technique which makes the calculation of the DFT much faster if a large number of harmonics is required. Equation (7.) can be expressed in a more compact form shown as follows,

$$\vec{v}(k\Omega_o) = \sum_{i=0}^{N-1} v(iT)\ e^{-jki\Omega_o T} \qquad (7.)$$

let $\vec{v}_k = \vec{v}(k\Omega_o)$

$v_i = v(iT)$

$\omega = e^{-j\Omega_o T}$

then $\vec{v}_k = \sum_{i=0}^{N-1} v_i \omega^{ik}$ (17.)

$$\begin{bmatrix} \vec{v}_1 \\ \cdot \\ \cdot \\ \cdot \\ \vec{v}_{N-1} \end{bmatrix} = \begin{bmatrix} \omega^0 \omega^0 \omega^0 \cdot \\ \omega^0 \omega^1 \omega^2 \cdot \\ \omega^0 \omega^2 \omega^4 \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \end{bmatrix} \begin{bmatrix} v_0 \\ \cdot \\ \cdot \\ \cdot \\ v_{N-1} \end{bmatrix} \tag{18.}$$

If all the $\vec{v}_k$ are calculated and N is relatively large, the FFT performs the calculation faster than the DFT by a factor of orders of magnitude. The difference in the number of computation is $N \log_2 N$ versus N^2 for FFT and DFT, respectively.

However, in protective relay applications, only a few $\vec{v}_k$ are needed and N is relatively small. In our investigation, N=32 samples per cycle is used and we are only interested in the fundamental component $\vec{v}_1$. As a result, there will be no obvious differences in the computation time between the DFT and FFT techniques. In addition, there is no limitation on the number of samples taken per cycle for the DFT algorithm. In contrast, the FFT requires N to be the integer power of 2. Hence, the DFT is chosen for our algorithm because it is easier to implement and the choice of the number of samples is flexible.

Chapter 3. The New Algorithm

3.1. Using Half of the Window Data

The objective of our new algorithm is to provide reliable information of the power system to the feedback control loop in a minimum possible time.

Based on the DFT theory, the new algorithm is modified to use only half of the window data in calculating the fundamental frequency phasor in order to reduce the computation time. That is, we use,

$$\vec{v_1}^k = \sum_{i=k-\frac{N}{2}+1}^{k} v_i \; e^{-j\frac{2\pi}{N}i} \quad \text{, with N=32} \tag{19.}$$

instead of $$\vec{v_1}^k = \sum_{i=k-N+1}^{k} v_i \; e^{-j\frac{2\pi}{N}i} \tag{10.}$$

A time–varying frequency voltage signal can be represented as,

$$v(t) = A\sin(2\pi ft + \psi(t)) \tag{20.}$$

where $\psi(t)$ is the phase angle of the signal in radians.

And $$\Delta f = \frac{d\psi(t)}{dt} \tag{21.}$$

The instantaneous frequency of the signal is,

$$f_{signal} = f_{fundamental} + \Delta f$$

and the instantaneous phase of the phasor is estimated by,

$$\psi_{est}^{k} = \tan^{-1}\left[\frac{\sum_{i=k-\frac{N}{2}+1}^{k} v_i \cos\left[\frac{2\pi i}{N}\right]}{\sum_{i=k-\frac{N}{2}+1}^{k} v_i \sin\left[\frac{2\pi i}{N}\right]}\right] \qquad (22.)$$

For a signal with a frequency deviation from 60 Hz (fundamental), the phase found by equation (22.) oscillates with large amplitude about a mean value which is close to the true phase. The magnitude of the oscillation is found to be double the fundamental frequency, regardless of the number of samples per cycle taken. Therefore, the error can be eliminated by averaging over one–half cycle. Another method is to average three calculations using three phases as the input signals.

3.2. Three Phase Voltage Input Signals

Three phase voltages are used in order to make the individual phase voltages less sensitive to variations than the method based on one phase. Three–phase voltages $\vec{v}_a$, $\vec{v}_b$ and $\vec{v}_c$ are 120^0 apart from each other. They can be used to calculate $\vec{v}_{1a}^{*}$ $\vec{v}_{1b}^{*}$ $\vec{v}_{1c}^{*}$ and $\psi_{est\,a}$, $\psi_{est\,b}$, $\psi_{est\,c}$ by using equations (19.) and (22.), respectively.

As mentioned above, the phase angle ψ_{est} oscillates about the mean value. However, it happens that $\psi_{est\,a}$, $\psi_{est\,b}$ and $\psi_{est\,c}$ are out of phase in such a way that an average of them is quite close to the mean value. That is,

$$\psi_{est\ avg} = \left[\frac{\psi_{est\ a} + \psi_{est\ b} + \psi_{est\ c}}{3}\right] \qquad (23.)$$

Three–phase voltages therefore provide more reliable phase sources than the single input phase. However, the algorithm will still work properly if only one or two voltages

sources are used as input data. For a one phase calculation, a running half cycle average is taken of all the phase calculation over the half cycle ending at the current time data point.

3.3. Rate of Change of Frequency Limiter

The typical frequency swing of a power system is at a rate of 0.1 to 1 Hz per second and, extreme cases may reach 10 Hz per second. Consequently, the new algorithm will limit the rate of change of the estimated frequency to be a maximum of 10 Hz per second. Any over–limit estimation will be considered an error due to a phase shift caused by a fault, and therefore the calculated frequency is held at the previously estimated value. This maximum expected value comes from the fact that the inertia constant H of typical generators is 2 to 9 [1]. Since H is half the time in seconds for a machine to go from zero speed (0 Hz) to full speed (60 Hz) with one–per unit torque applied, it follows that a 100% load drop would result in a frequency change rate of $\frac{60}{2H}$ Hz/sec . Manitoba Hydro's lowest H is about 3, giving a rate of 10 Hz/sec maximum.

3.4. Computer Software

All the programs in this investigation were written in the standard C computer language, and all the simulations and tests were done on the Unix based Sun Workstation. Moreover, Xuniplot is used to plot all the test results.

3.5. Program Flow–Chart

Figure (7.) and Figure (8.) on the following pages show the computer program flow charts given in description and equations, respectively. The program performs the modified

DFT calculation and frequency estimation every n samples and n can be set to any positive integer. For our investigation, N is set to 32. The setting of n also depends on many factors such as: how fast the clock is, what the memory size is on the board, and what the maximum delay allowed, etc.

Starting from the top of the chart , a new set of data is gathered with new input and the previous input stored in the memory. The digital filter is then applied to the voltage signals. However, the digital filter is optional for our testing. It is used to filter the test signals with higher order harmonic and noise components in order to meet the Nyquist criterion as discussed in a previous chapter. After the filtering, the modified DFT algorithm is used to find the complex phasors and then the phase angles. Comparing the previous and present phase angles, we can find the rate of change of frequency. If the rate of change of frequency is not greater than 10 Hz/sec, as checked by the limiter, a new frequency is estimated. The previous estimated frequency is kept if the limit is violated.

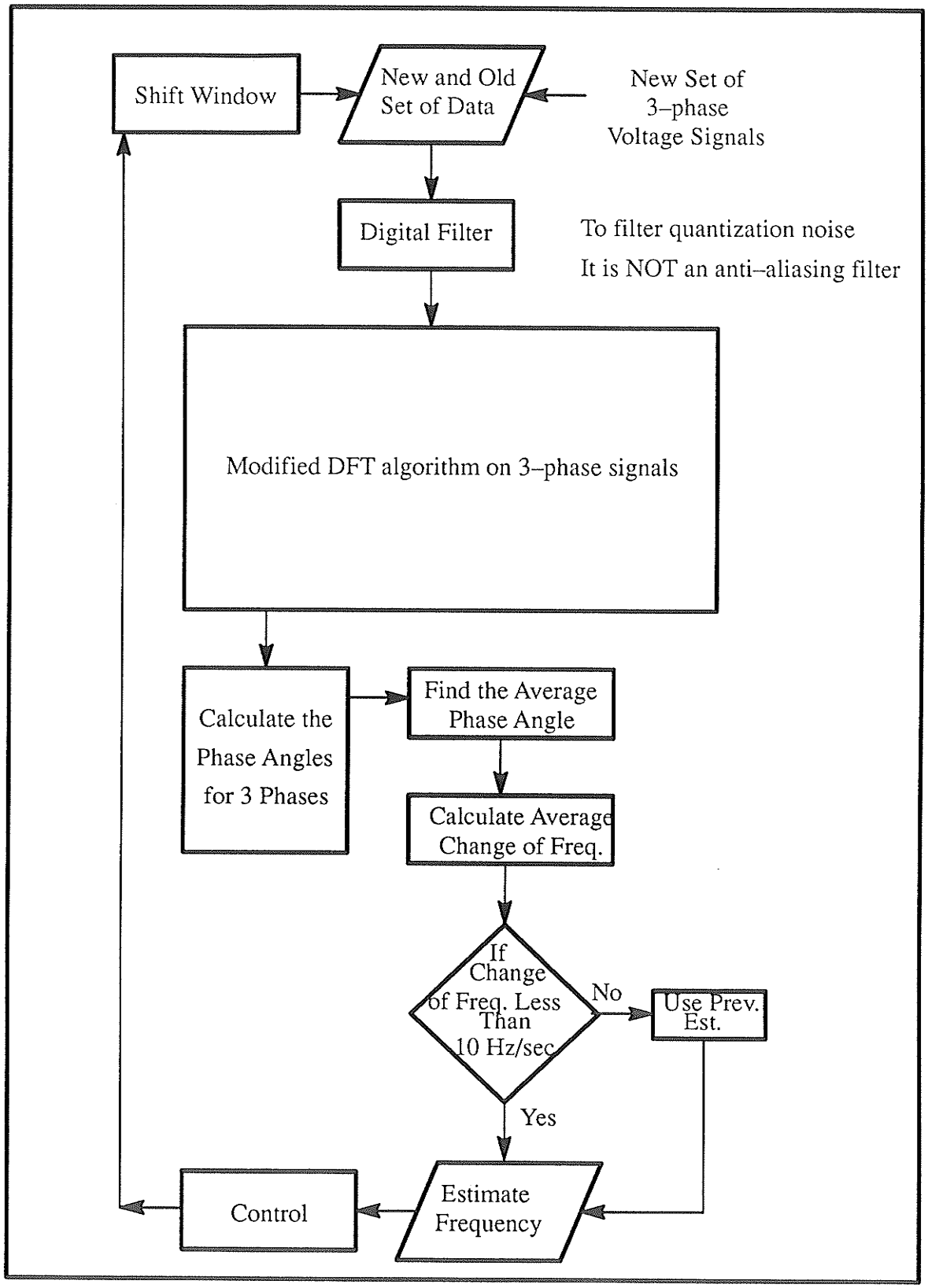


Figure 7. Flow Chart in Description

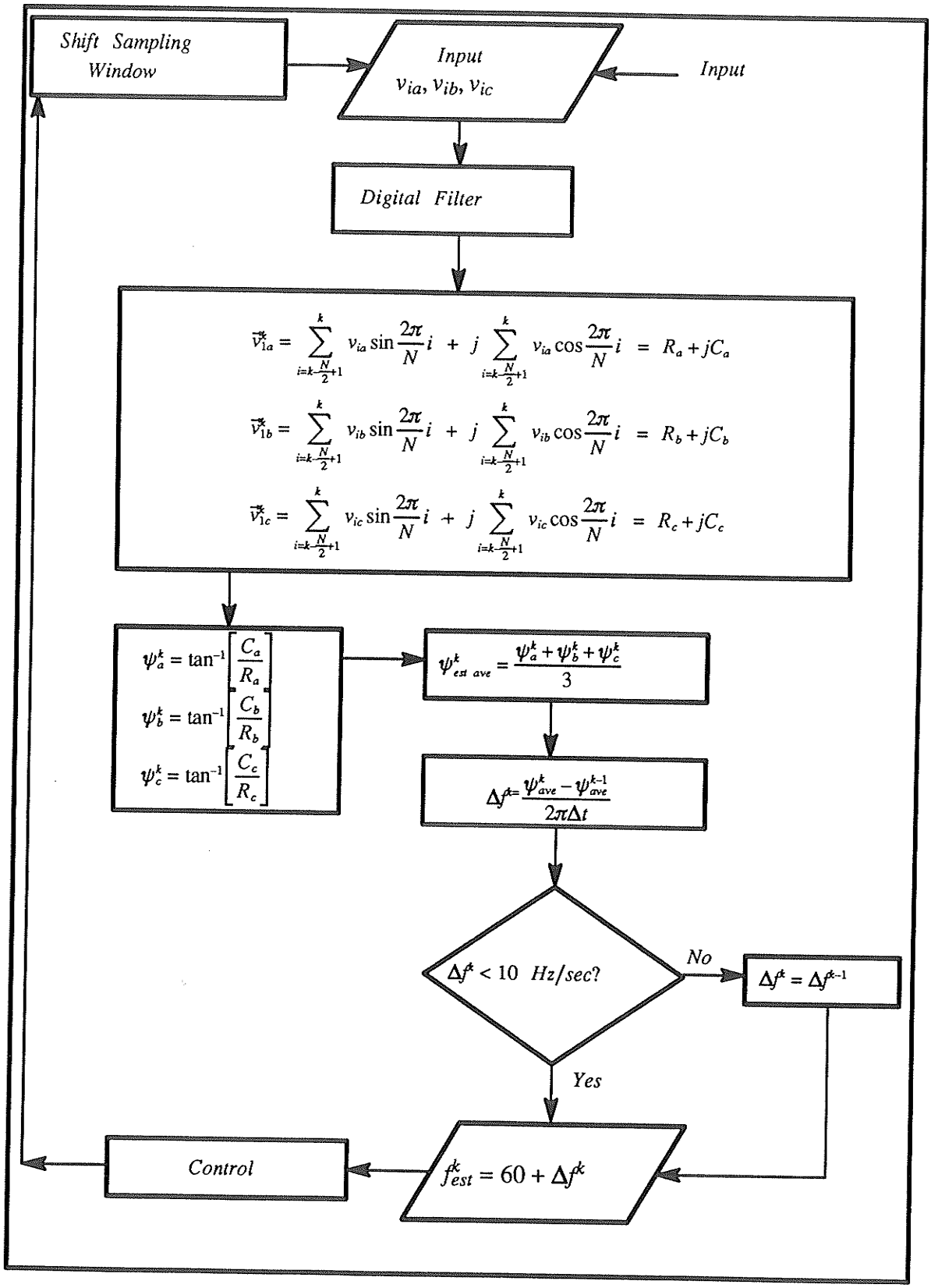


Figure 8. Flow Chart in Equations

Chapter 4. Typical Test Signals

All the test signals are chosen to simulate typical situations for a power system. Our algorithm aims to track the system frequency when the system is in a power swing condition, as well as in the steady state. The program should not have any problem in dealing with signals corrupted by, for example, harmonics and noise, provided that the signals are filtered with an anti–aliasing filter. Furthermore, the algorithm should also be immune to distortion caused by a sudden jump in the voltage magnitude and the phase change of the signals.

There are many advantages in choosing phase voltages, rather than currents, as the input signals. One obvious advantage is that the voltages would not be cut off in case of the open circuit of the power system.

Six different cases are investigated in this thesis:

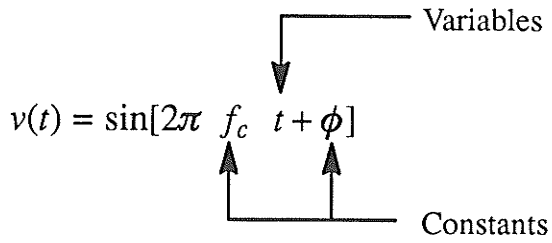
1. Frequency swing signals.
2. Frequency swing signals with 3rd harmonics overlapping at various phase angles.
3. Frequency swing signals with 3rd and 5th harmonics overlapping signals in phase.
4. Frequency swing signals with high frequency noise.
5. Steady state (60 Hz) signals with 50% magnitude drop.
6. Steady state (60 Hz) signals with 90 degrees phase shift over one cycle.

4.1. Frequency Swing Signal

A power system undergoes transient frequency swings when there is an unbalance between the generation and the load. Control equipment must adjust accordingly, for example load shedding, in order to protect the system from collapse.

4.1.1. How to Define a Signal with Time–varying Frequency

A steady–state sinusoidal voltage signal is defined as,

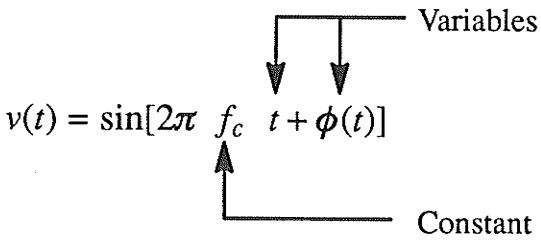


(24.)

where f_c is the carrier frequency which is 60 Hz for our investigation,

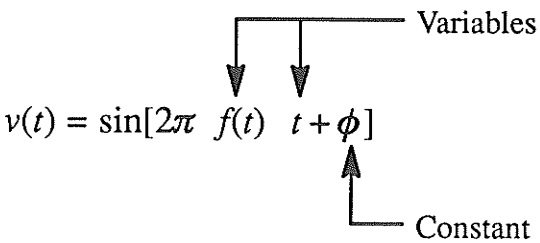
and ϕ is a constant phase angle.

However, a signal with time–varying frequency, that is , a swing condition would be defined by



(25.)

Note that it is NOT correct to use



(26.)

In other words, the variable phase has the effect of producing a variable frequency signal [9].

$$\Delta f(t) = \frac{d\phi}{dt} \tag{27.}$$

Manitoba Hydro requires that the relay must be able to track a frequency change at a maximum of 10 Hz per second. Thus the variable frequency is represented by

$$f(t) = 60 - 0.8\sin(4\pi t) \tag{28.}$$

$$\frac{df}{dt} = 0.8(4\pi)\cos(4\pi t) \tag{29.}$$

$$\left[\frac{df}{dt}\right]_{\max} = 0.8(4\pi) = 10.05 Hz/sec \tag{30.}$$

and

$$\psi(t) = 2\pi \int f(t)dt \tag{31.}$$

$$\psi(t) = 0.4\cos(4\pi t) + constant \tag{32.}$$

The corresponding voltage signal is therefore given by;

$$v(t) = sin[2\pi 60t + 0.4cos(4\pi t) + \phi_b] \tag{33.}$$

where ϕ_b is an arbitrary constant phase angle

The time span for each testing signal will last for 0.55 seconds and it is set from –0.05 to 0.5 seconds. For frequency swing signals, the frequency varies between the times 0.0 to 0.375 seconds, and is constant during the remaining time. A signal will be as follows with respect to time,

$$v(t) = \sin(2\pi 60t + \phi_a] \qquad t < 0 \qquad (34.)$$

$$v(t) = sin(2\pi 60t + 0.4\cos(4\pi t) + \phi_b) \qquad 0 \leq t \leq 0.0375 \text{ sec} \qquad (35.)$$

$$v(t) = sin(2\pi 60t + 2\pi(0.8)t + \phi_c) \qquad t > 0.0375 \text{ sec} \qquad (36.)$$

ϕ_b in equation(35.) and ϕ_c in equation (36.) are chosen such that the phase is continuous at the transition between the three functions. Otherwise, a step phase change could occur and result in an instant impulse frequency. The frequency of the signal after 0.375 seconds is kept constant at about 60.8 Hz, in order to investigate if the algorithm still works at a non–60 Hz signal. Figure (9.) shows a plot of frequency of the signal against the time. Figure (10.) shows the rate of change of frequency of Figure (9.).

For three phase signals, ϕ_a, ϕ_b and ϕ_c will have the same form as equations (34.), (35.) and (36.) with 120 degrees difference between them. Figure (11.) shows the 3–phase voltage used as the inputs over a period from –0.02 to 0.03 seconds. Per Unit (pu) is used for all the voltage signals.

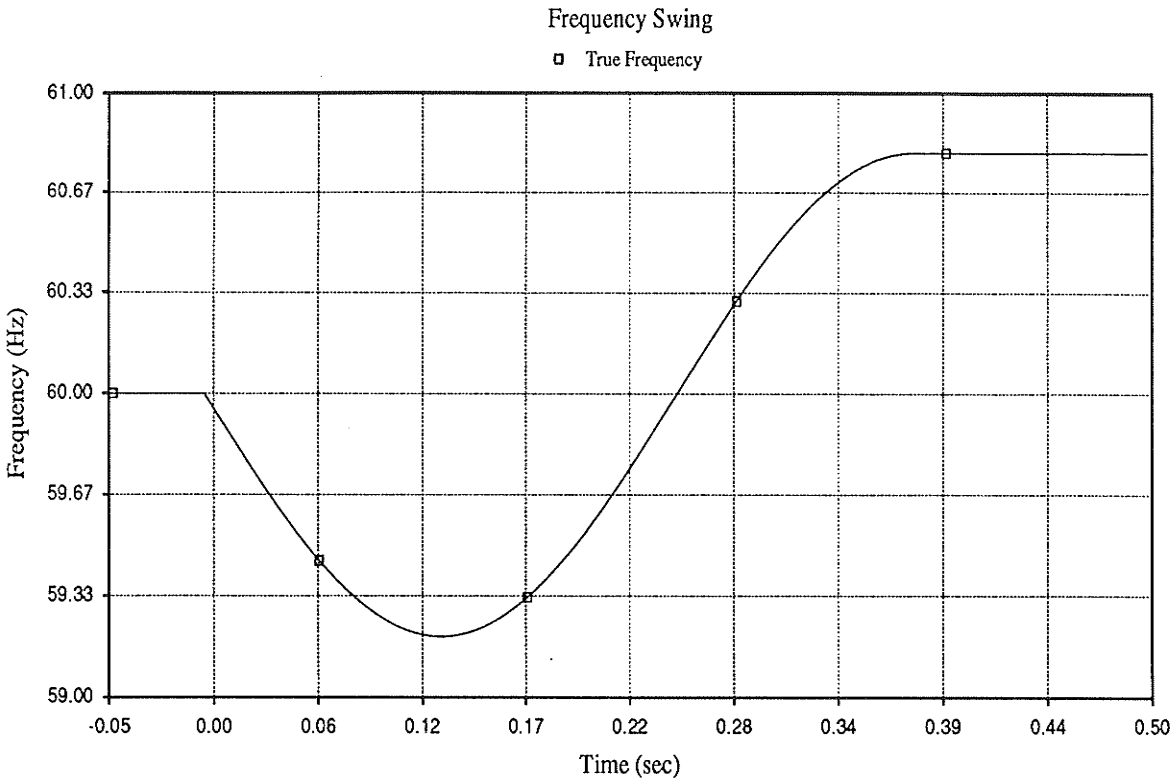


Figure 9. Frequency Swing

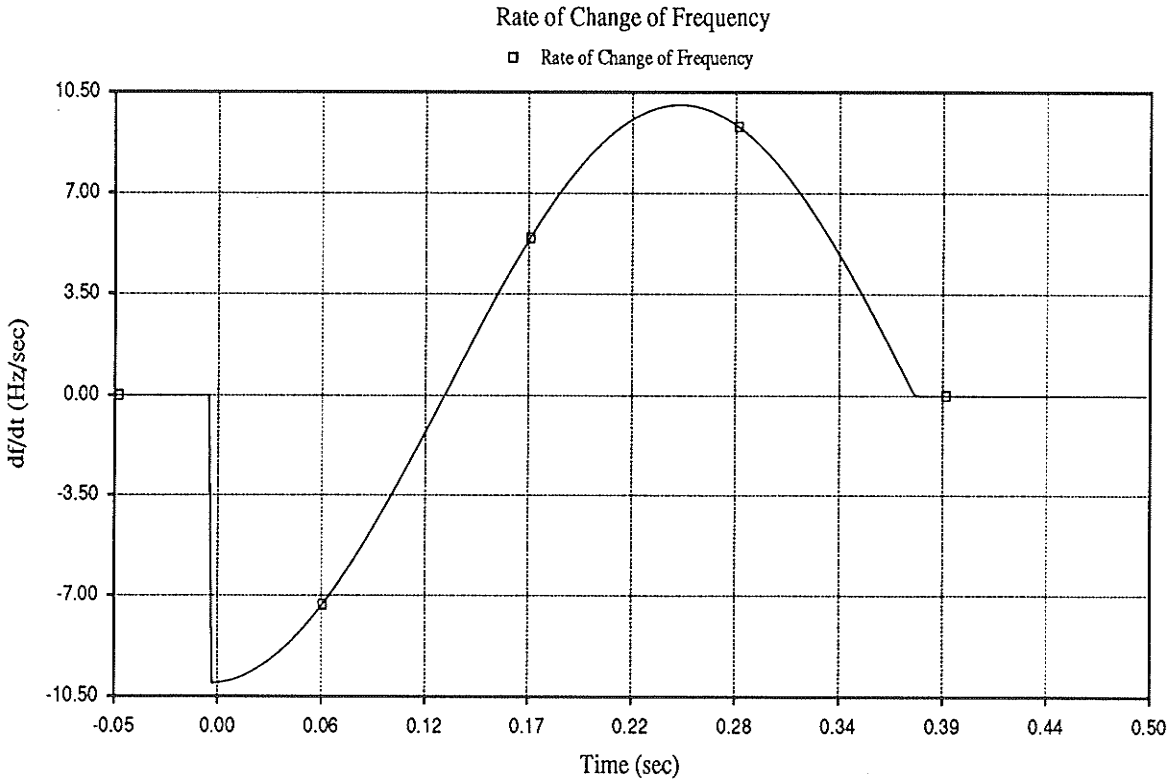


Figure 10. Rate of Change of Frequency

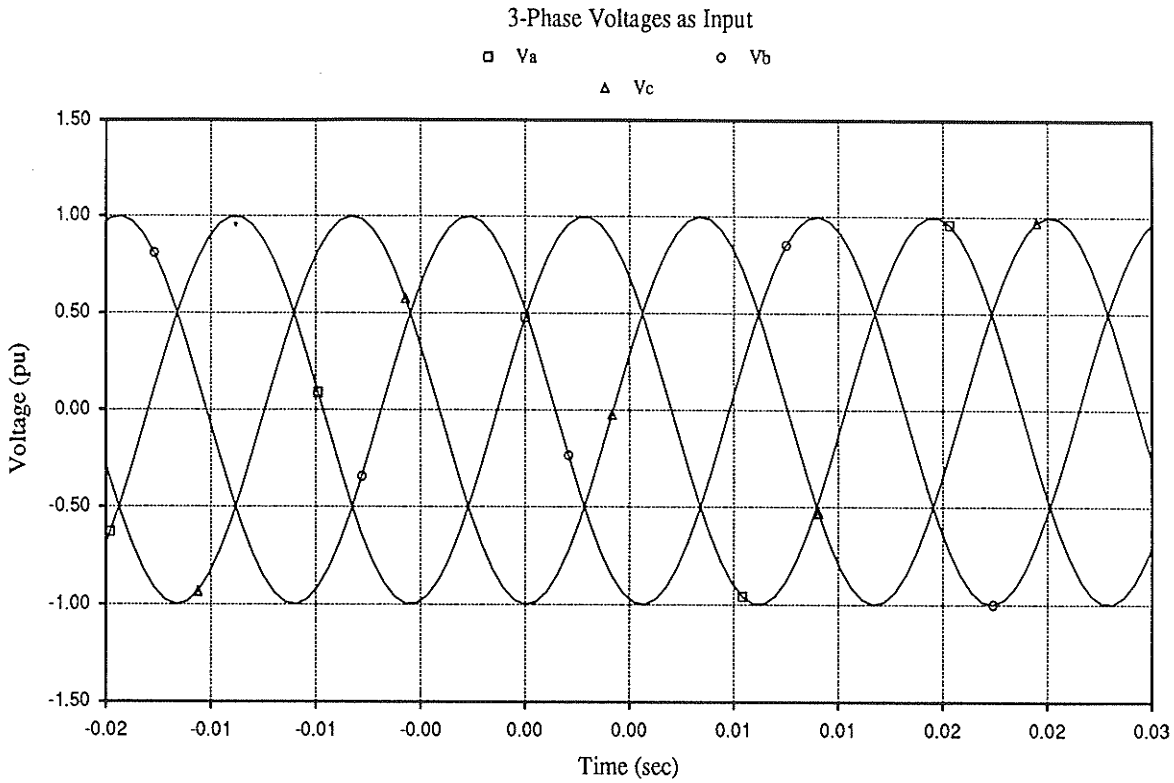


Figure 11. Three–phase Voltage Signals

4.2. Frequency Swing Signals with 10% 3rd Harmonic (in magnitude) at Various Overlapping Angles.

Harmonics are distortions to the voltage and current waveforms from their normal sinusoidal shape. A 60 Hz sine wave is generated at the power generating stations and distributed to a large number of residential and industrial loads. Certain types of loads distort the fundamental (60 Hz) by injecting additional signals of various magnitudes and frequencies. Harmonics are also sinusoidal in shape but their frequencies are integer multiples of the fundamental signals. However, the magnitudes of the harmonics normally decrease with increasing frequency. Hence, only the first few orders usually need to be considered in examining the effects of harmonics on power system component or equipment. The presence of harmonics creates a steady state problem which may lead to some serious effects on computer systems, capacitor banks (overloaded), communications lines and

protective relay performance. There are basically two sources of harmonics generated from power system components namely, solid state devices such as thyristors and transformers operating with non–linear voltage and current characteristics.

For this part of the testing, we will add 3rd harmonics to the fundamental wave. The magnitude of the 3rd harmonics is chosen as 10% of the fundamental, and the overlapping angles (γ) between two signals are as follows:

1. $\gamma = 0°$
2. $\gamma = 15°$
3. $\gamma = 45°$
4. $\gamma = 60°$
5. $\gamma = 90°$

Figures (12.) shows the resulting voltage waveforms at 90 degree overlapping angle over the period of –0.02 to 0.03 seconds.

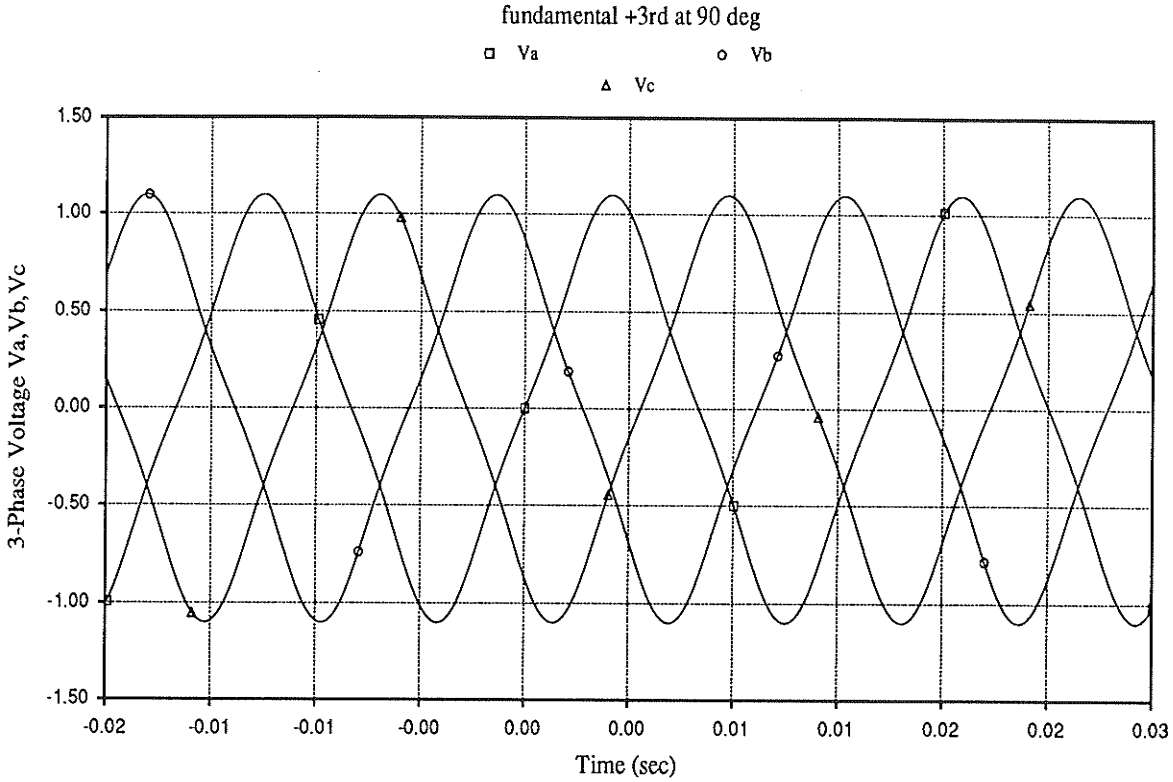


Figure 12. Voltage Signals with 10% 3rd Harmonics at γ=90°

4.3. Frequency Swing Signal with 10% 3rd and 1% 5th Harmonics in Phase with the Fundamental

The fundamental voltage signals are injected with 10% 3rd and 1% 5th harmonics at a angle $\gamma = 0°$. The waveform of the corrupted signal will be as follows,

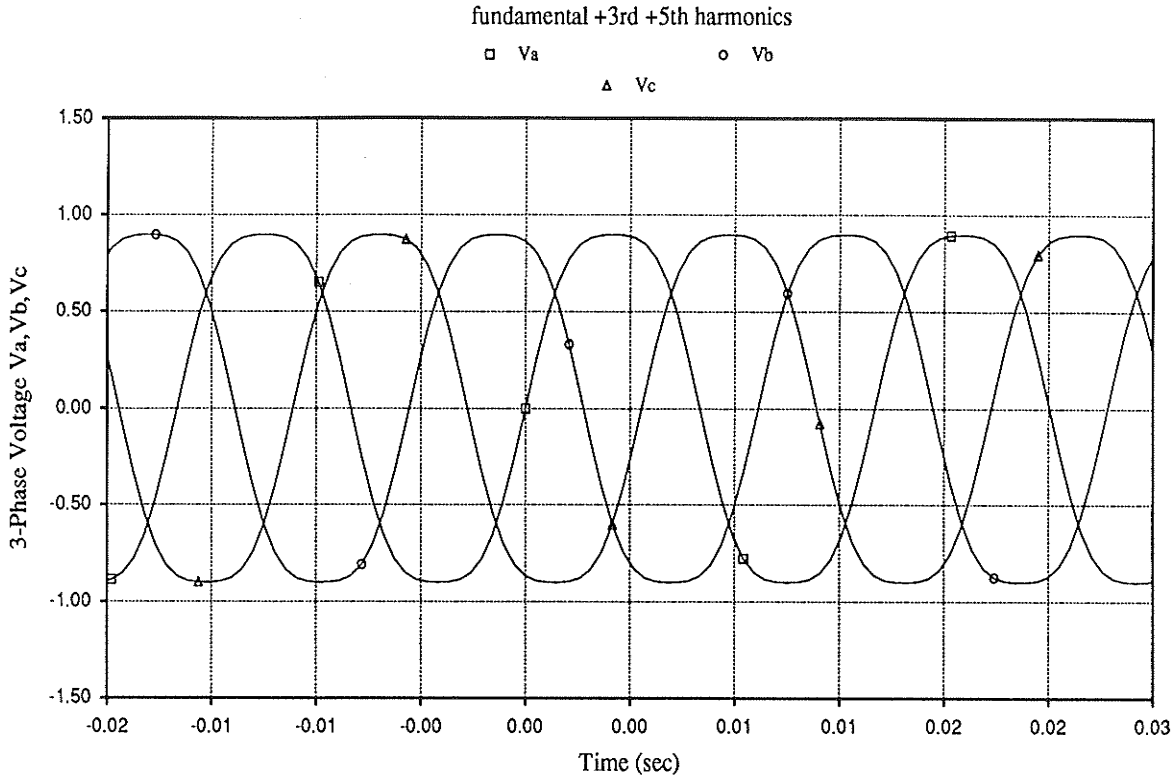


Figure 13. Voltage Signals with 10% 3rd and 1% 5th Harmonics at $\gamma = 0°$

4.4. Frequency Swing Signal with Gaussian Distributed High Frequency Random Noise

Random noise can be picked up by the voltage signals anywhere along the transmission line and at the power system components. It can also be introduces "quantization" noise in the A/D converter. As mentioned previously, all signals we are testing have to meet the Nyquist Criterion to prevent aliasing. We assume all the test signals have been filtered with anti–aliasing filters. Therefore, no anti–aliasing filtering is installed in our simulations. Referring to Figure (14.), the simulation starts after the analog–digital (A/D) converter. For this particular test, a random noise (for example, quantization error noise) is added to signals coming out of the A/D converter. Also, a low pass filter is used to eliminate the high frequency components in order to minimize its effect on the DFT algorithm.

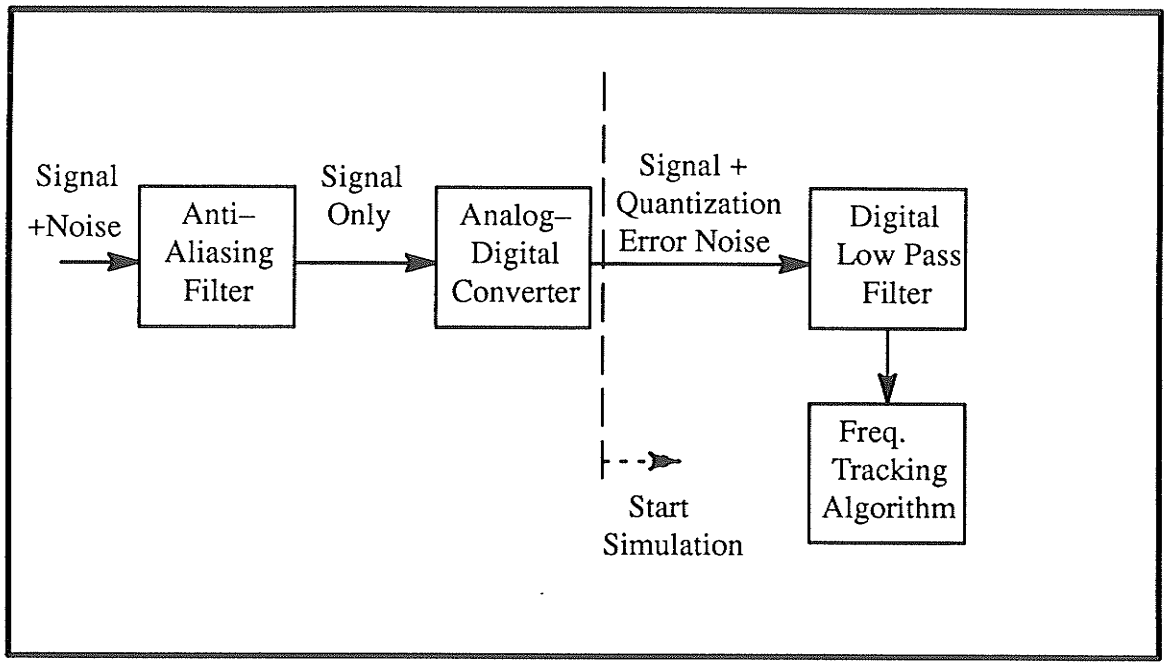


Figure 14. Configuration of Computer Program Simulation

4.4.1. Gaussian Distributed Random Noises

One popular type of noise is the Gaussian distributed random noise, which has a bell–shape probability distribution[10]. A typical Gaussian distributed probability graph is shown in figure (15.). The variance σ is chosen to be 0.01 meaning that 68% of the random numbers generated are between –0.01 and 0.01, and 95% between –0.02 and 0.02. The random noise is thus about 1% to 2% of the fundamental signals in magnitude, and Figure (16.) shows how a random noise waveform may appear;

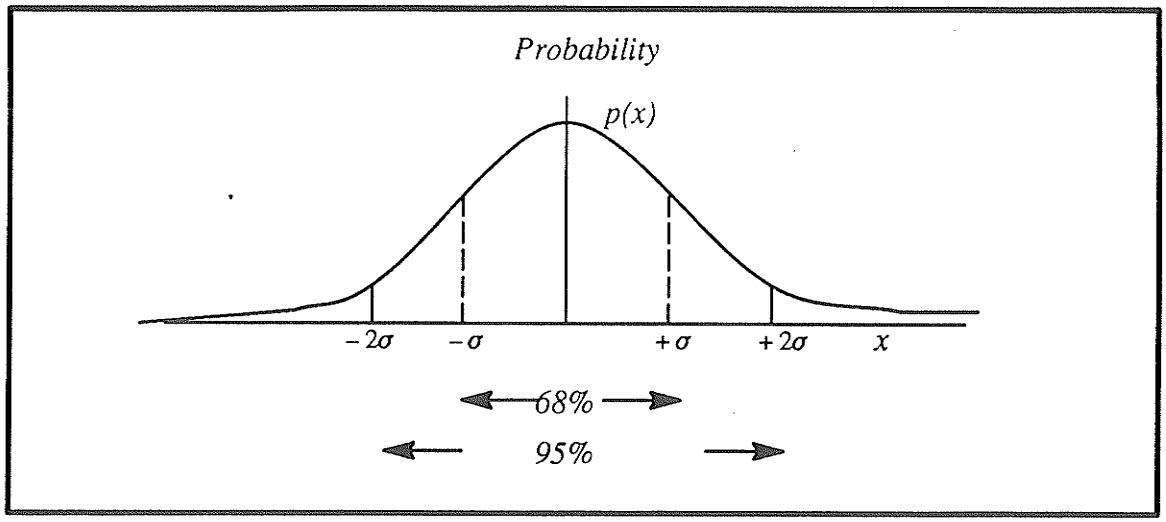


Figure 15. Gaussian Distributed Probability Graph

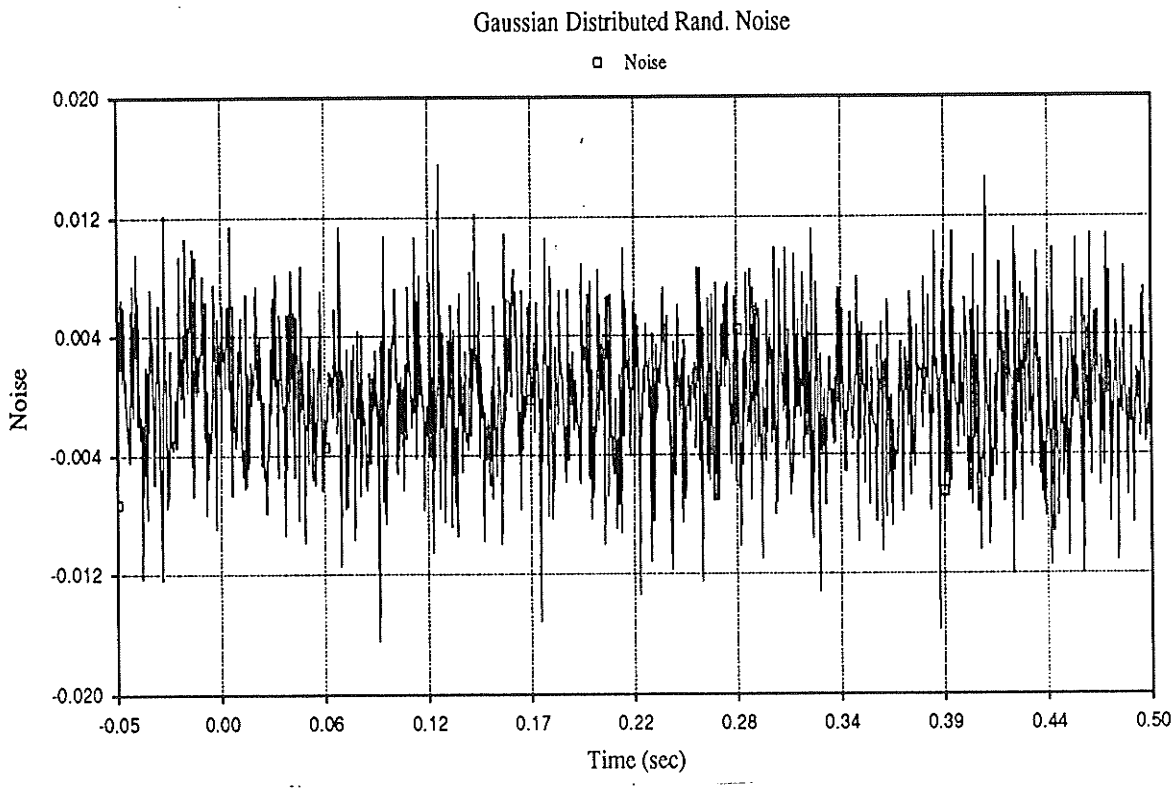


Figure 16. Computer Generated Gaussian Distributed Random Noise Waveform

4.4.2. Low Pass Digital Filters

The digital filter [11] is constructed from software to eliminate undesired quantization noise harmonics. [An analog filter is constructed from analog hardware, and is usually installed before the analog–digital converter to eliminate higher harmonic components according to the sampling theorem.] In our investigation, the digital filter is used only to remove a quantization noise introduced by the A/D converter. We do not use this filter for the purpose of anti–aliasing.

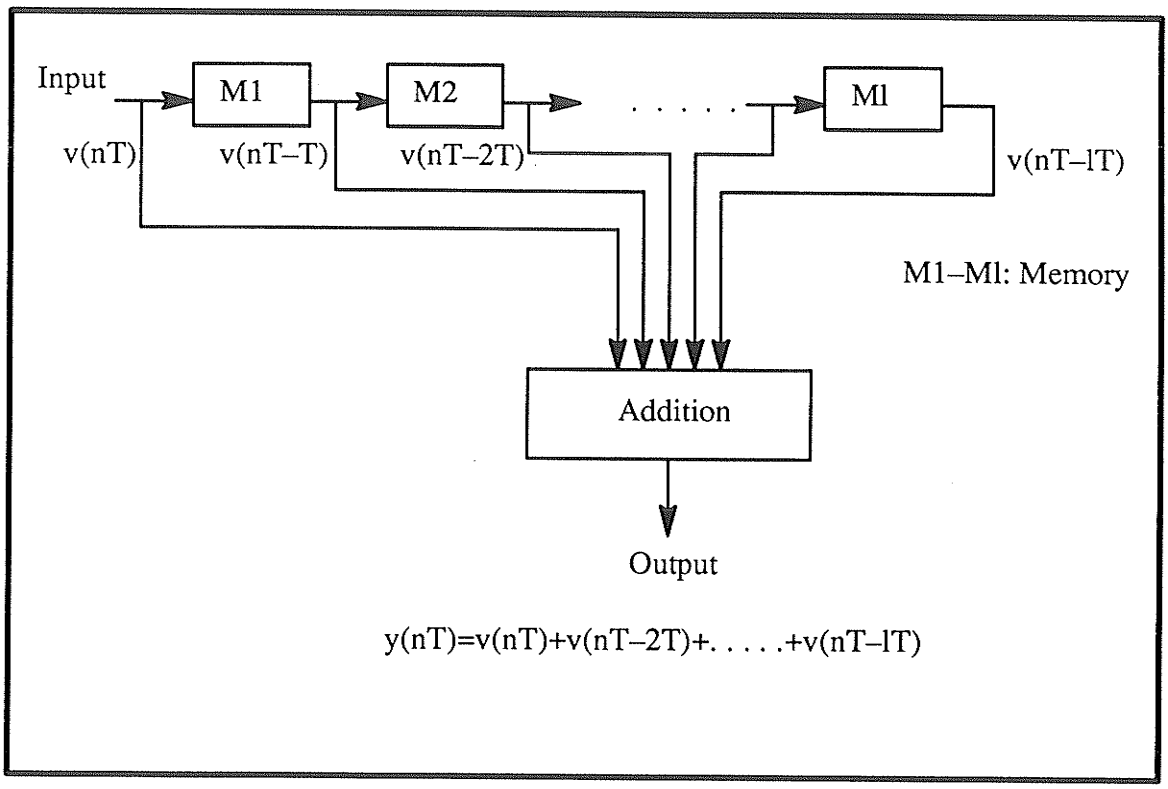


Figure 17. Block Diagram of a Low–Pass (Integral) Filter.

The output of the integral filter is;

$$y(nT) = v(nT) + v(nT - T) + \cdot \cdot \cdot \cdot + v(nT - lT) \tag{37.}$$

Taking the Z–Transform of equation (37.), we get,

$$y(Z) = v(Z) \quad (1 + Z^{-1} + \cdot \cdot \cdot \cdot + Z^{-l})$$

$$y(Z) = v(Z) \left[\frac{1 - Z^{-1-l}}{1 - Z^{-1}} \right] \tag{38.}$$

The transfer function is,

$$T_A = \frac{y(Z)}{v(Z)}$$

$$T_A = \left[\frac{1 - Z^{-1-l}}{1 - Z^{-1}}\right] \tag{39.}$$

And the gain is [11],

$$G_A = \left[\frac{\sin\left[\frac{(1+l)\omega T}{2}\right]}{\sin\left[\frac{\omega T}{2}\right]}\right] \tag{40.}$$

According to equation (40.), the frequency response characteristics of the integral filter are given by,

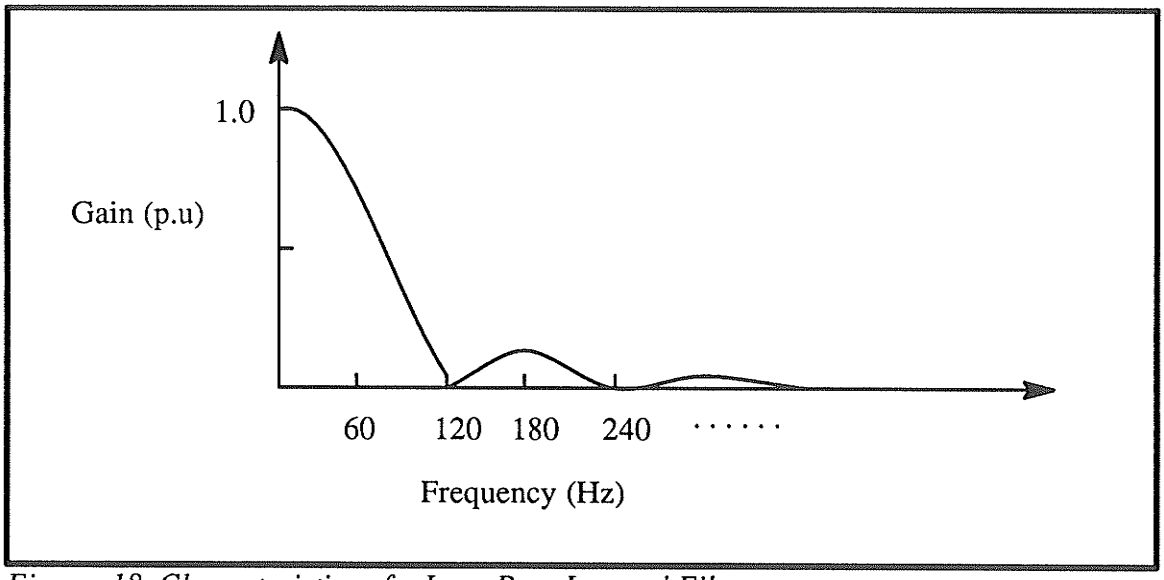


Figure 18. Characteristics of a Low–Pass Integral Filter

This type of integral filter can eliminate higher harmonics of frequency given by,

$$f = \frac{h}{(l+1)T} \qquad (h = 1, 2, \cdot\ \cdot\ \cdot) \tag{41.}$$

where T is the period of the sampling frequency which is the inverse of 60xN Hz,

Using N=32 and l=15, equation (40.) becomes;

$$G_A = \left[\frac{\sin(\frac{\pi}{2} h)}{\sin(\frac{\pi}{16})} \right] \tag{42.}$$

And equation (41.) is given by,

$$f = 120h \qquad (h = 1, 2, \cdot \cdot \cdot) \tag{43.}$$

Specifically, the filter eliminates all the harmonics of integer multiplies of 120 Hz and its attenuation is proportional to the inverse of the frequency. This filter is particularly useful to our algorithm as the fundamental signal is preserved after the filtering.

4.5. Fundamental Voltage Signal (60 Hz) with 50% Sudden Drop in Magnitude

4.5.1. Voltage Signal Before and During the Fault

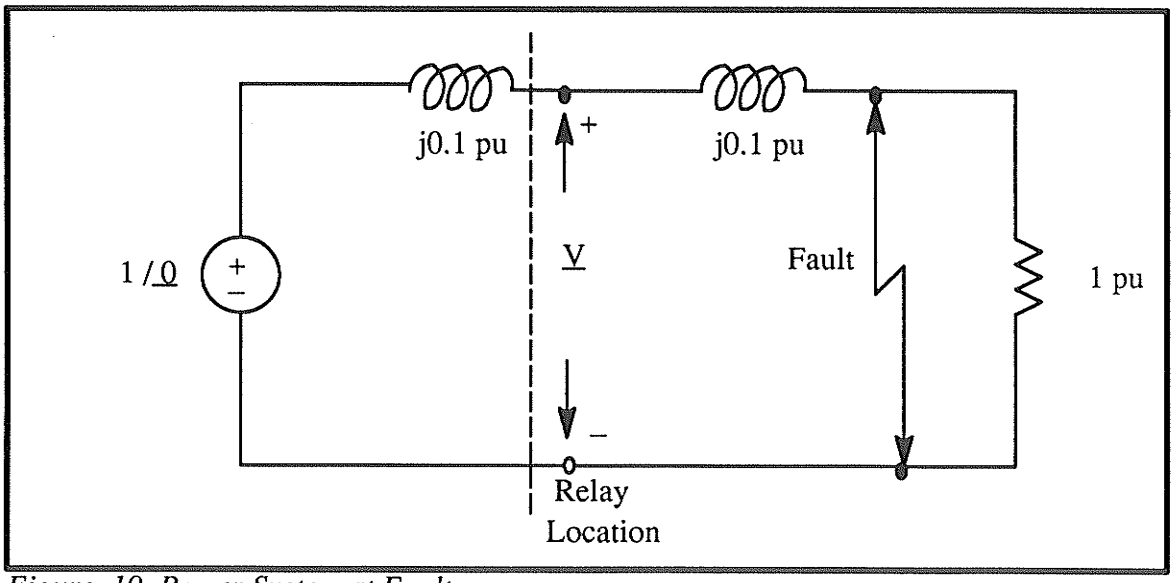


Figure 19. Power System at Fault

Figure (19.) shows a simplified circuit diagram of a faulted power system. The voltage signal before and during the fault are;

before the fault: $\underline{V} = \frac{1 + j0.1}{1 + j0.2} \quad 1/\underline{0^0} \quad = 0.98/\underline{-5.6^0}$

during the fault: $\overline{V} = \frac{j0.1}{j0.2} \quad 1/\underline{0^0} \quad = 0.5/\underline{0^0}$

The magnitude and the phase angle of the voltage change during the fault.

All the previous four cases dealt with frequency swing signals containing either harmonics or noise. Now the algorithm is tested for its response to a sudden magnitude drop in the fundamental voltage waveform (which is 60 Hz). Figure (19.) shows how a magnitude drop may occur. The frequency of the signal is kept constant at 60 Hz and the magnitude of the voltage is dropped to half of its original at time=0 seconds, as shown in the following figure.

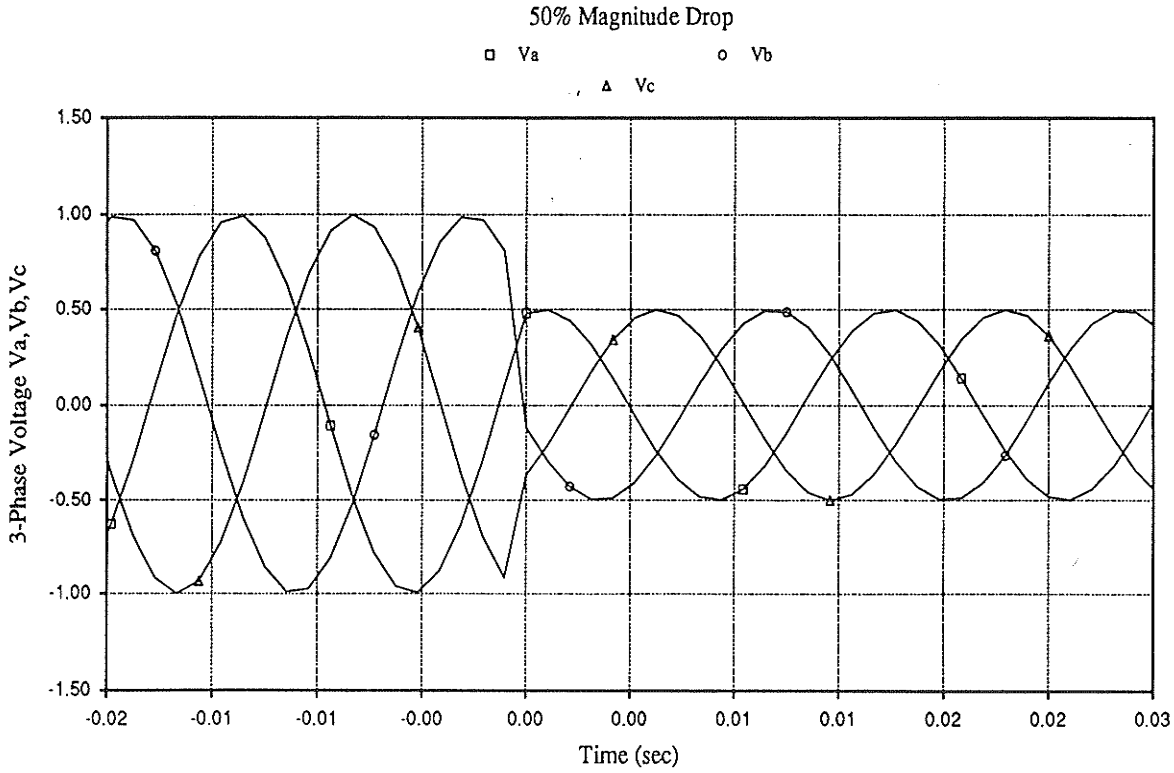


Figure 20. Fundamental Voltage Signals with 50% Drop in Magnitude

4.6. Fundamental Signal That Change Phase by –90 Degrees Over A Period of One Cycle

Figure (19.) illustrates how a sudden phase shift may occur, over say one cycle. A more severe example is chosen here: a 90^0 phase shift over a period of one cycle.

$$v(t) = sin(2\pi 60t) \qquad \text{for } t \leq 0 \text{ sec} \qquad f = 60Hz \quad (44.)$$

$$v(t) = sin(2\pi 60t - 30\pi t) \qquad \text{for } 0 \leq t \leq \frac{1}{60} \text{ sec} \qquad f = 45Hz \quad (45.)$$

$$v(t) = sin\left(2\pi 60t - \frac{\pi}{2}\right) \qquad \text{for } t > \frac{1}{60} \text{ sec} \qquad f = 60Hz \quad (46.)$$

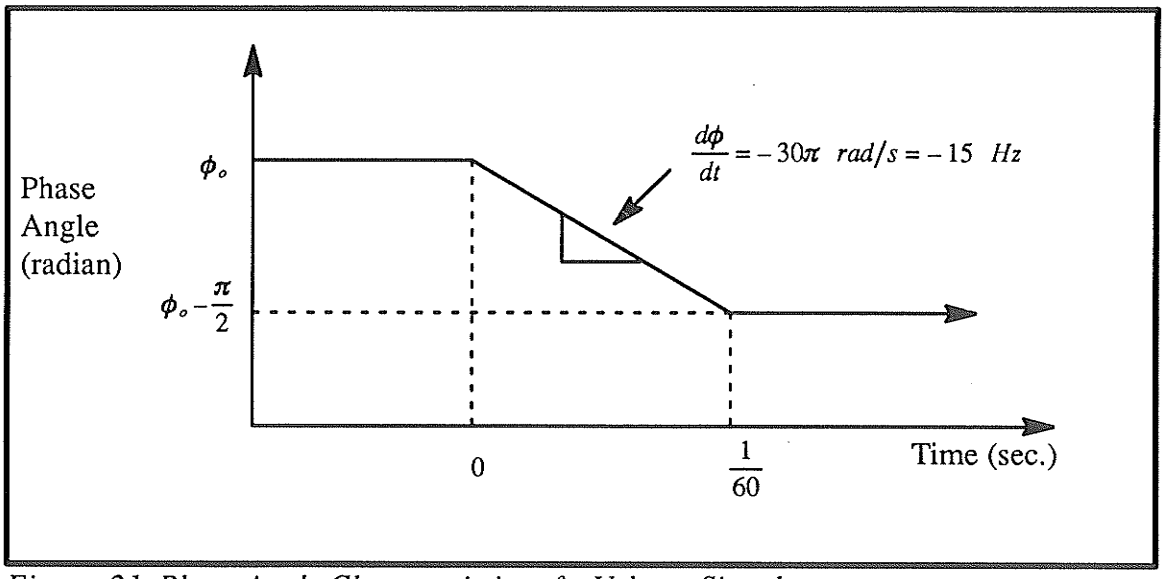


Figure 21. Phase Angle Characteristics of a Voltage Signal

A change in the phase angle shifts the frequency from the fundamental frequency (60 Hz) to 45 Hz for a period of one cycle. Figure (22.) illustrates the change in frequency between time=0 to time=$\frac{1}{60}$ seconds. The signal at time=$\frac{1}{60}$ is 90^o out of phase with the original waveforms.

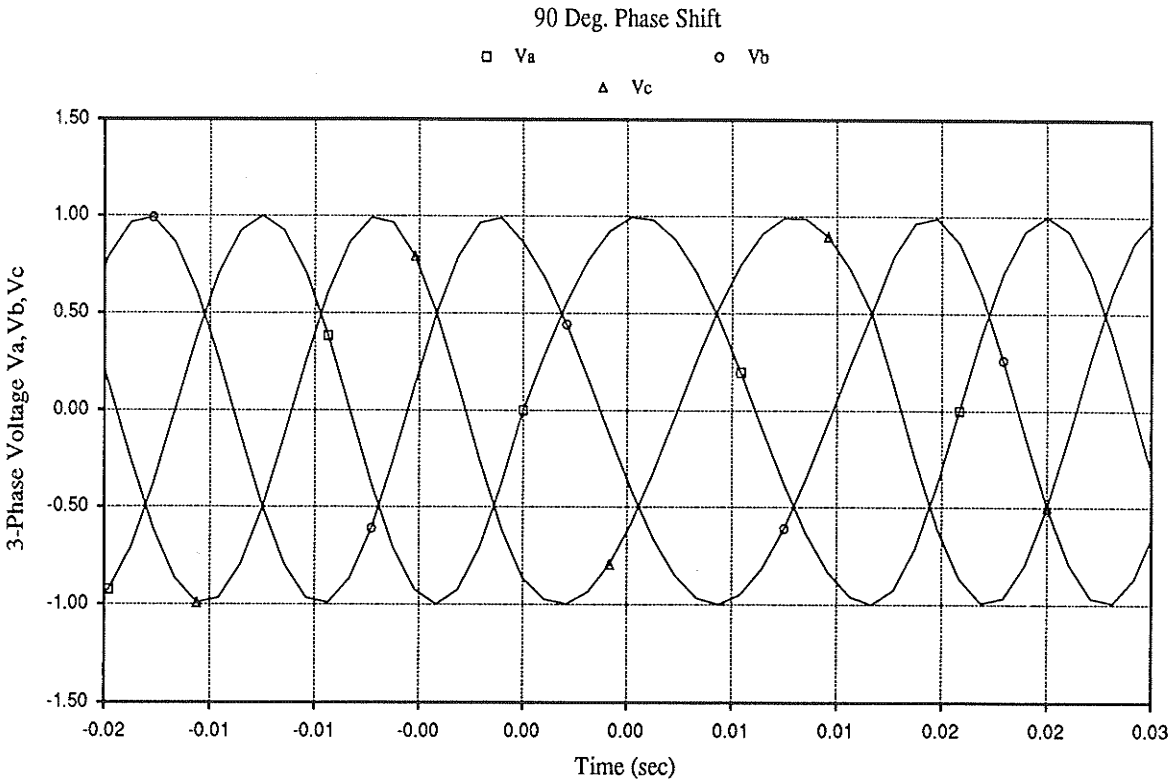


Figure 22. Fundamental Voltage Signal that Shift –90 degrees over one cycle

Chapter 5. Test Results

5.1. Frequency Swing Signal

Appendix (1) contains computer program which estimates the frequency of the input signal under the power swing condition. We use N=32 samples per cycle as our sampling rate for all of our testings. Figure (23.) shows the frequency response of the algorithm. The Y–axis is the frequency of all of the 3–phase voltages. The X–axis is the program simulation time from –0.05 to 0.5 seconds. As observed from the graphs, the algorithm responds quite accurately under the ideal power swing situation. The maximum delay time is only about 4 to 5 milliseconds, or a quarter of a cycle, during the period of the power swing. The response is extremely accurate during the period of 60–Hz and non–60–Hz constant signals. The frequency of the 3–phase voltage signals is constant at 60 Hz before time=0, and swings from 60 to 59.2 Hz, then to 60.8 Hz where it stays at a constant. Figure (25.) shows the true and estimated rate of change of frequency. It should also be noted that the delay would be bigger if anti–aliasing filters are included.

5.1.1. Three–Phase Versus One–Phase

In order to show that using 3–phase inputs is better and faster than only 1–phase, figure (24.) is included for comparison with figure (23.) . The result shows the delay time would be about double if only the 1–phase voltage signal is used as the input. However, the delay time of half of a cycle, is still quite small and acceptable.

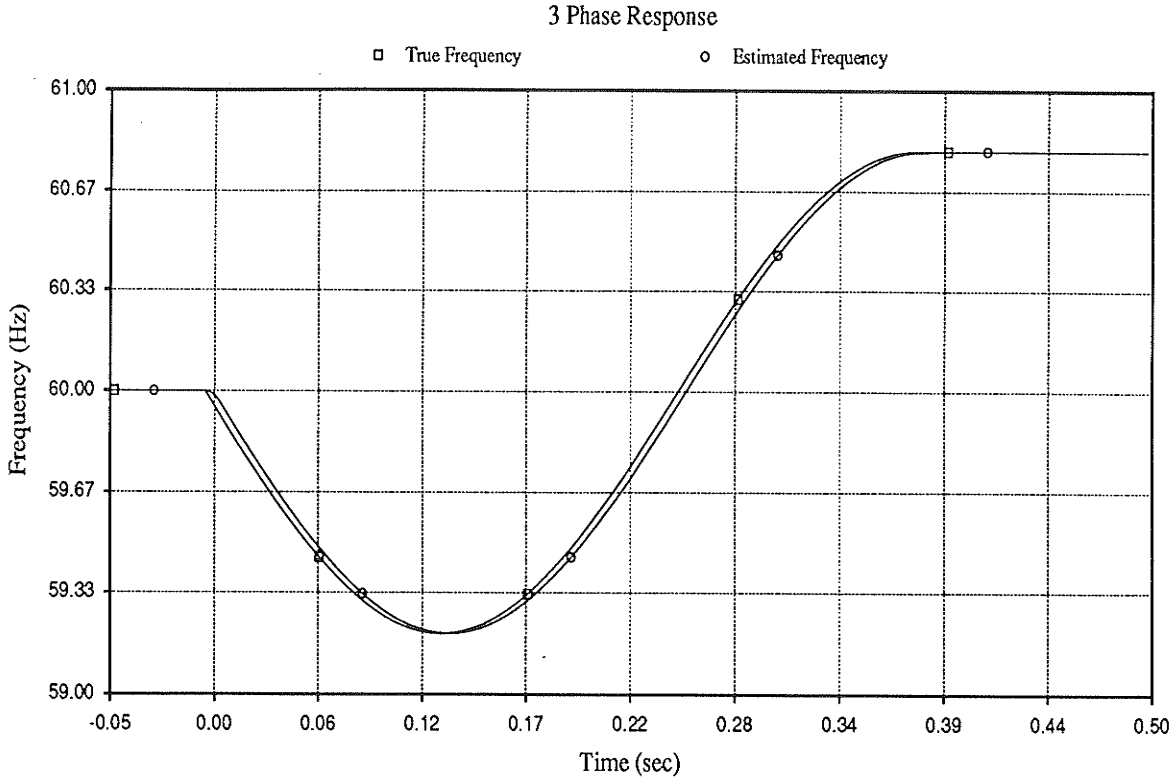


Figure 23. Result: Frequency Response of Pure Power Swing Three–Phase Signals

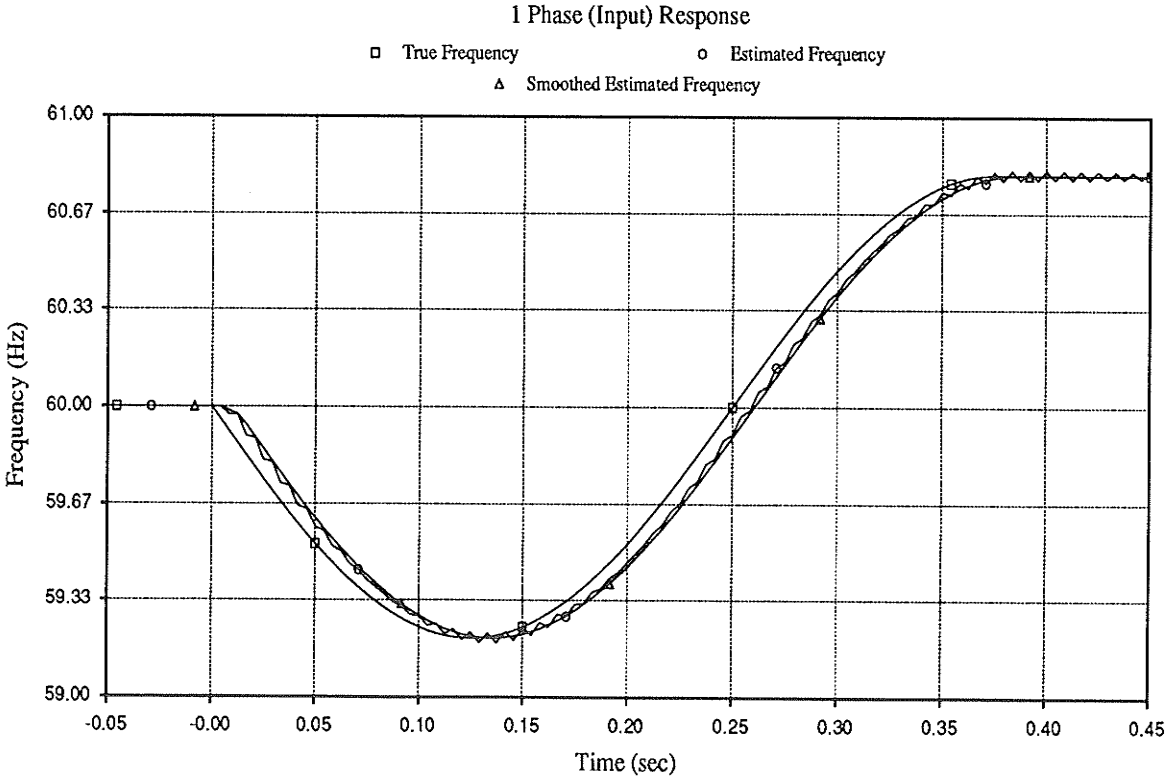


Figure 24. Results: One–phase Response

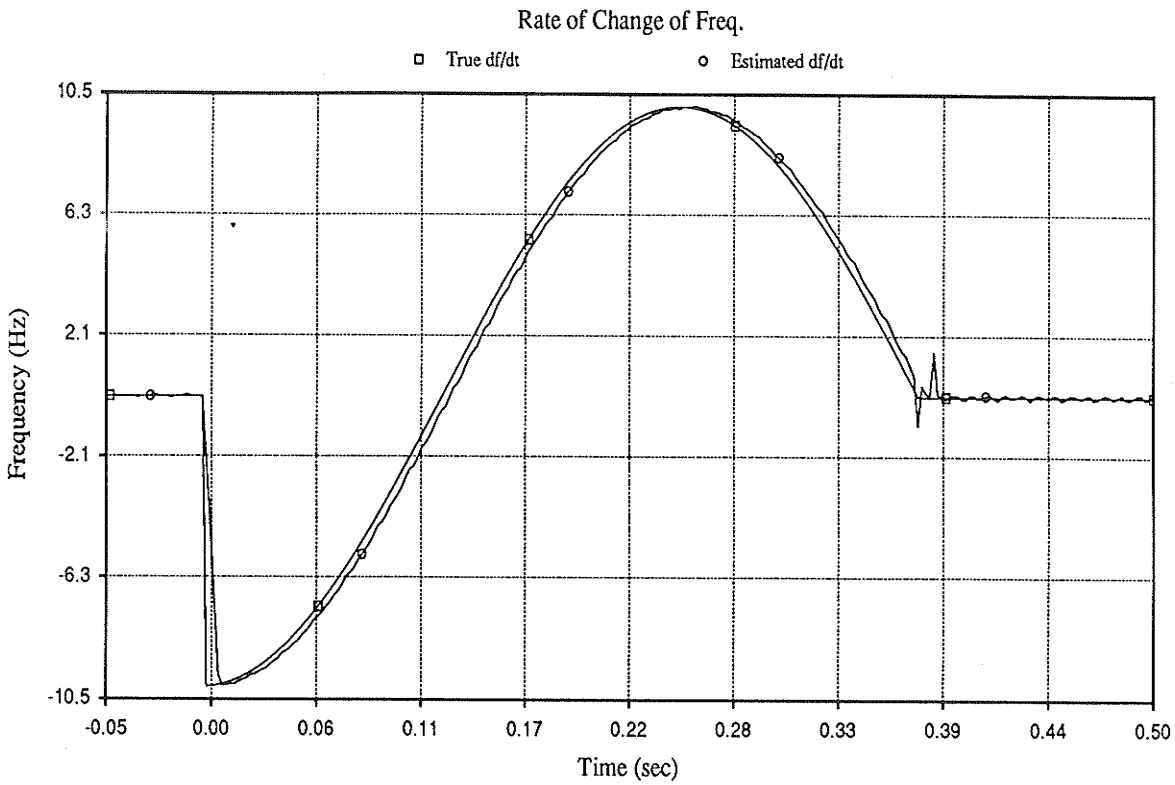


Figure 25. Result: Rate of Change of Frequency

5.2. Frequency Swing Signal with 10% 3rd Harmonic (in magnitude) at Various Overlapping Angles.

Figures (26.) to (35.) show the results of frequency responses for signals with 10% 3rd harmonics at different overlapping angles. The figures on the top of each page are the 3–phase voltages over the period of –0.02 to 0.03 seconds. The figures on the bottom of each page are the frequency responses. The computer program of the frequency relay algorithm is listed in Appendix (2).

By observing the voltages signals, it is obvious the waveform has been changed significantly as the overlapping angle $\gamma = 0^{\circ}$ to $\gamma = 90^{\circ}$. However, the distorted input waveforms have negligible effects on the frequency response. The algorithm is thus proven to be insensitive to the low order harmonics.

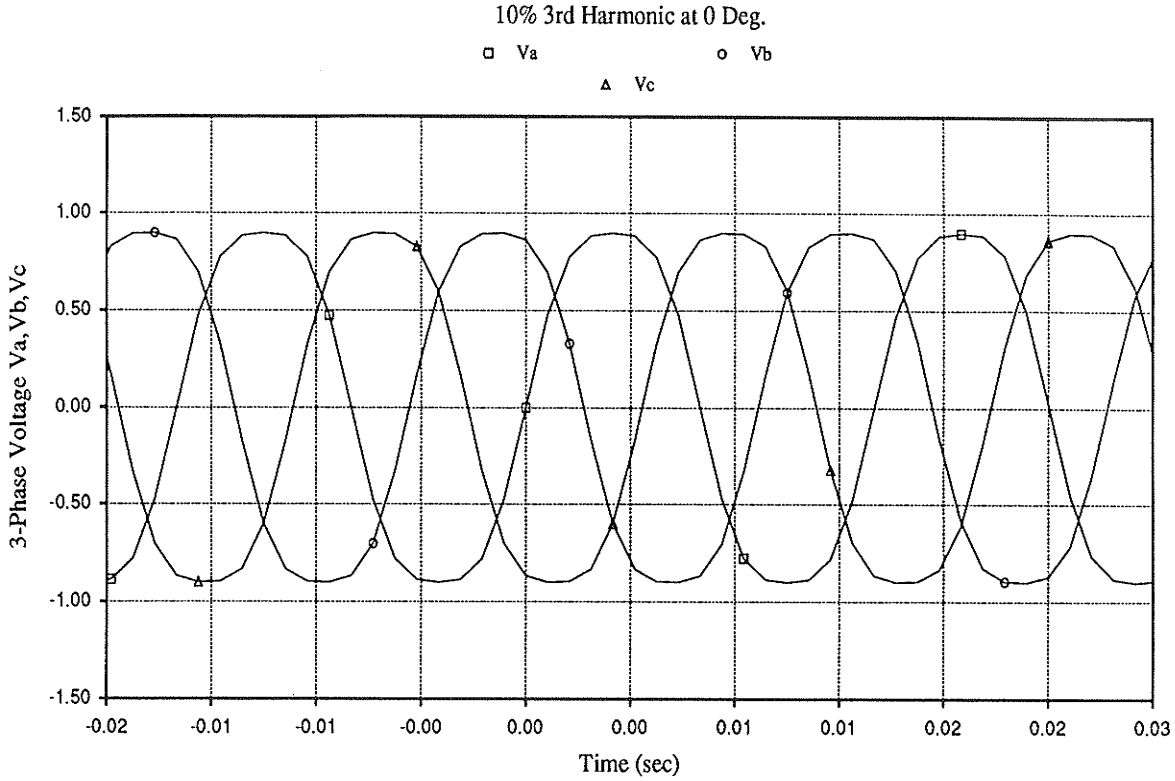


Figure 26. Voltage Signals with 10% 3rd Harmonic at γ=0°

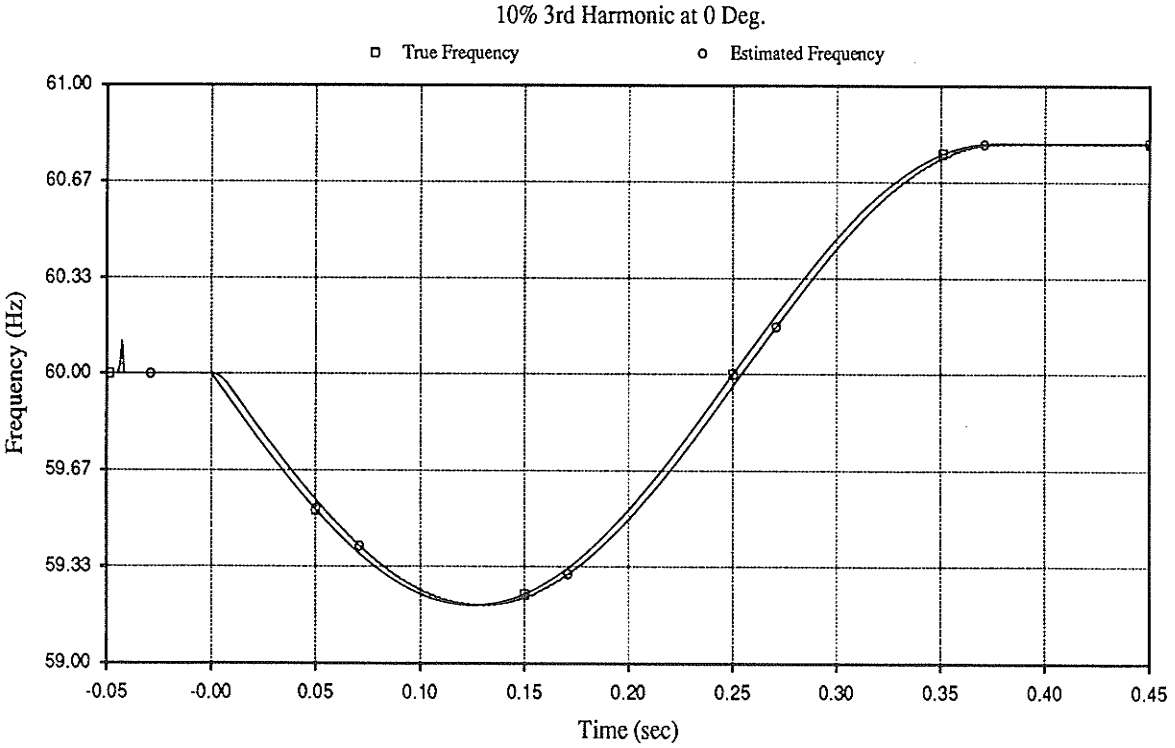


Figure 27. Result: 10% 3rd Harmonic at γ=0°

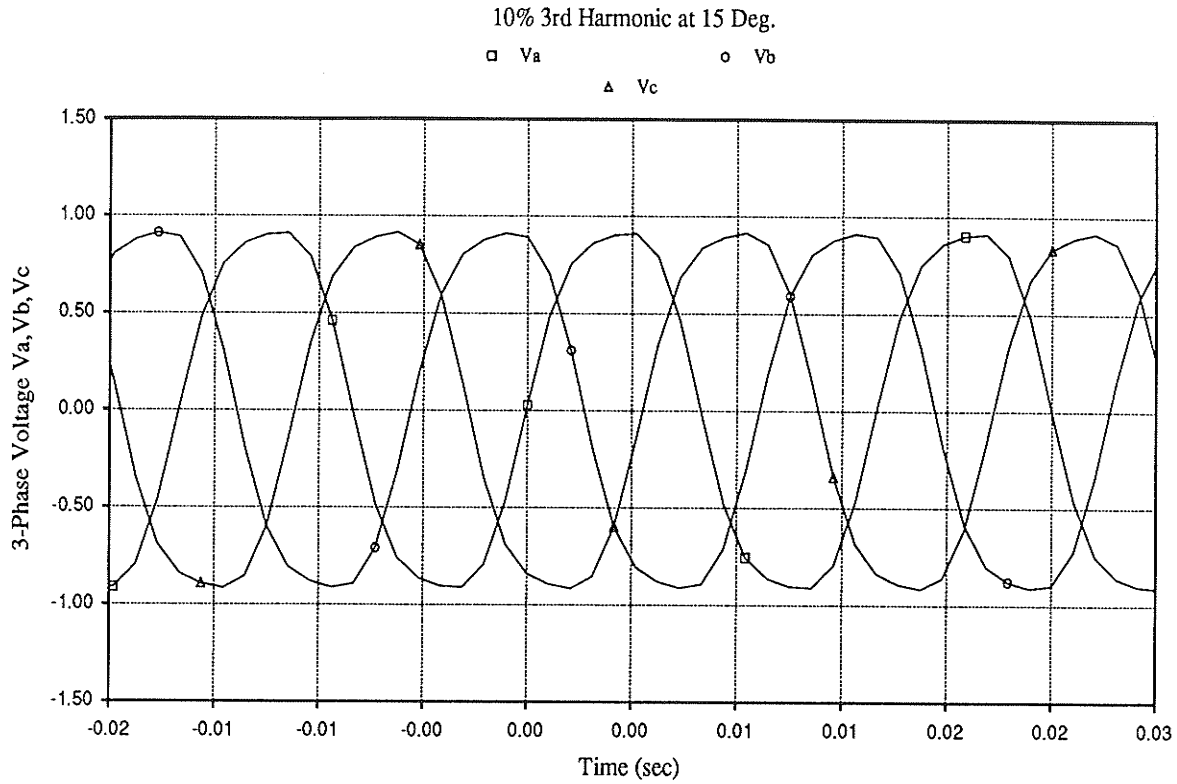


Figure 28. Voltage Signals with 10% 3rd Harmonic at γ=15°

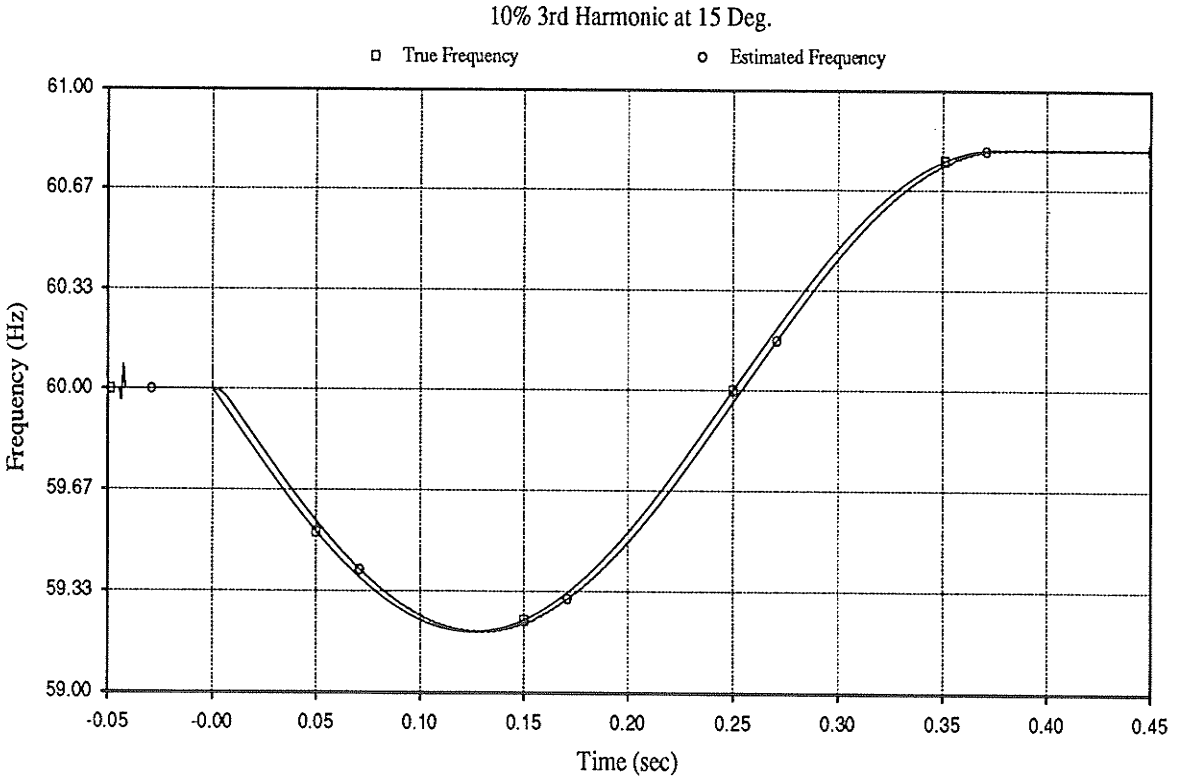


Figure 29. Result: 10% 3rd Harmonic at γ=15°

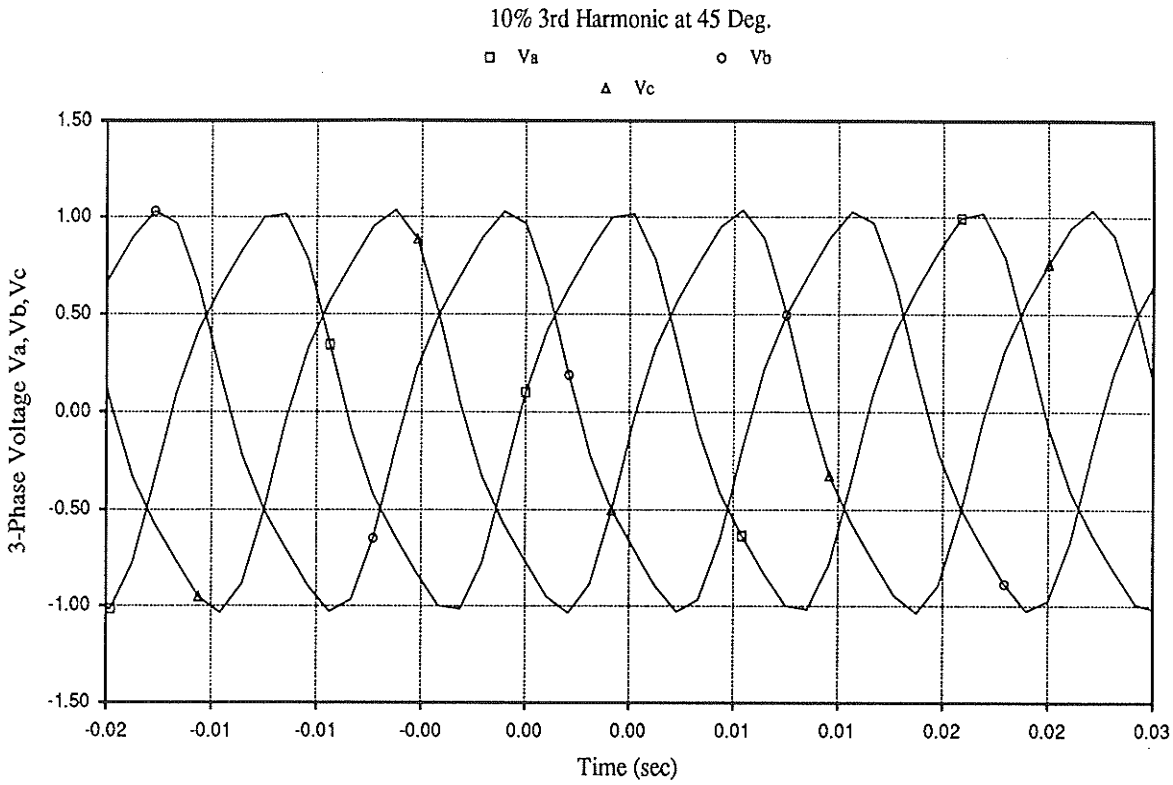


Figure 30. Voltage Signals with 10% 3rd Harmonic at γ=45°

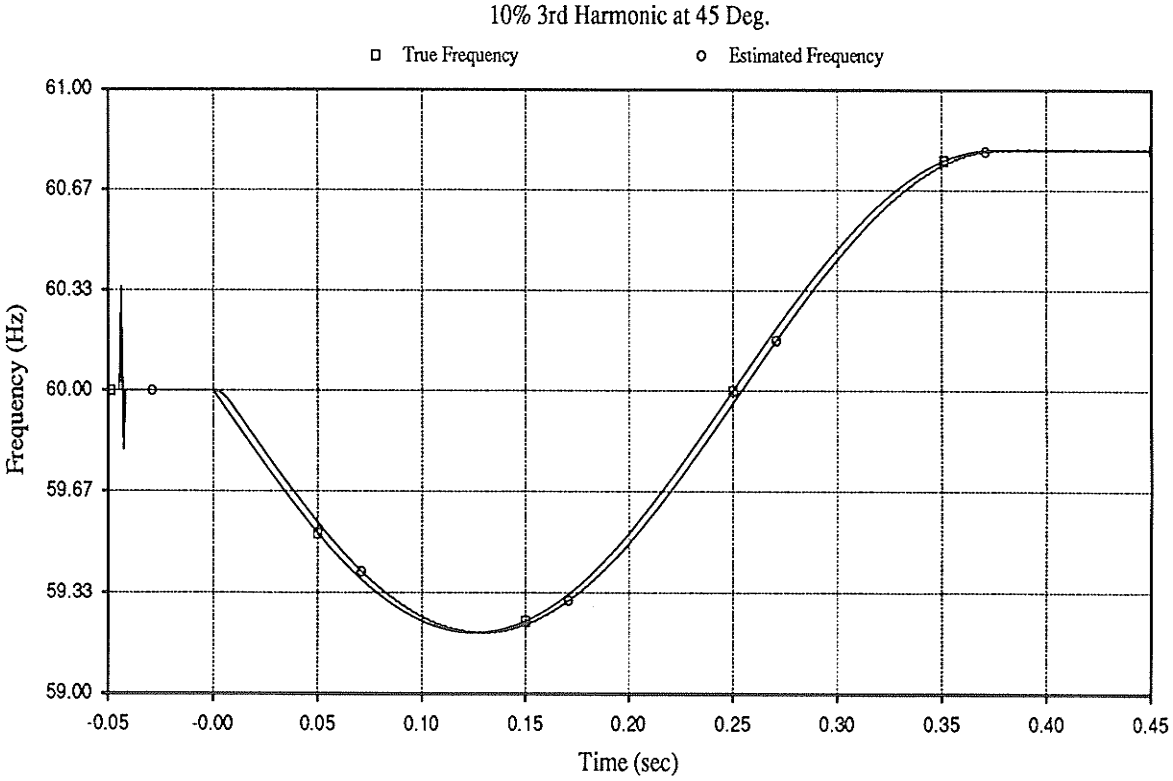


Figure 31. Result: 10% 3rd Harmonic at γ=45°

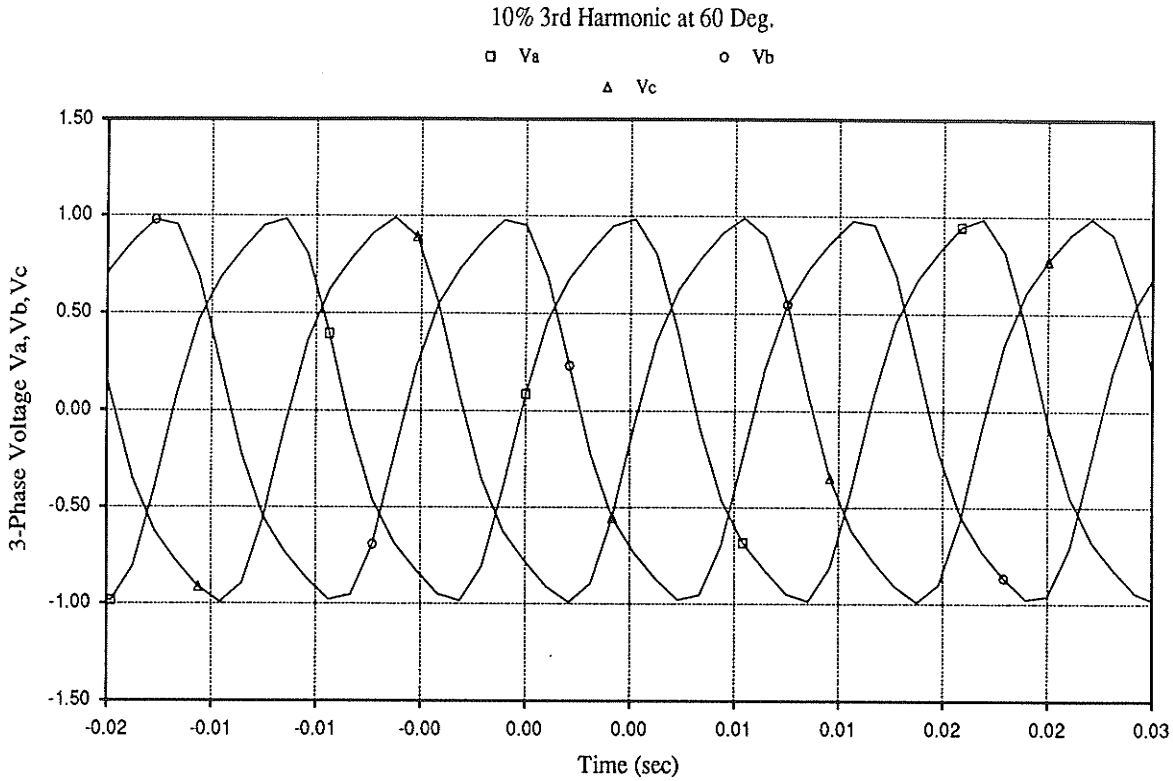


Figure 32. Voltage Signals with 10% 3rd Harmonic at γ=60°

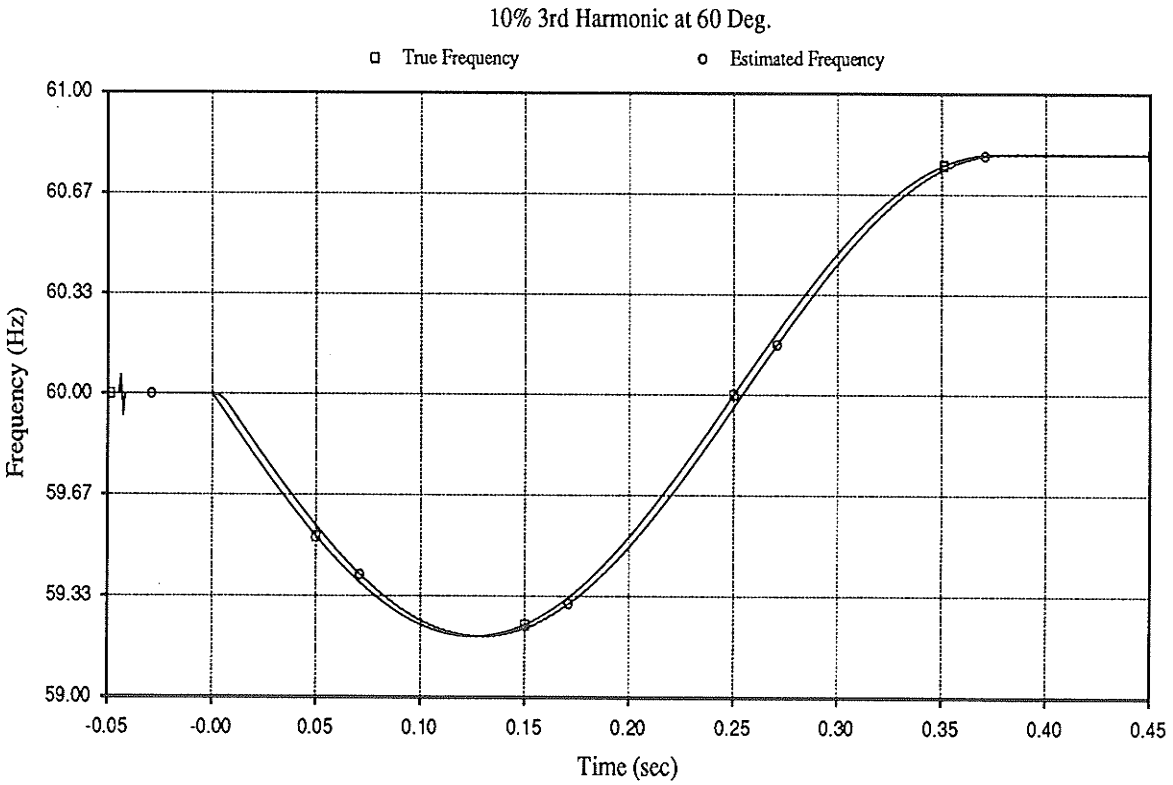


Figure 33. Result: 10% 3rd Harmonic at γ=60°

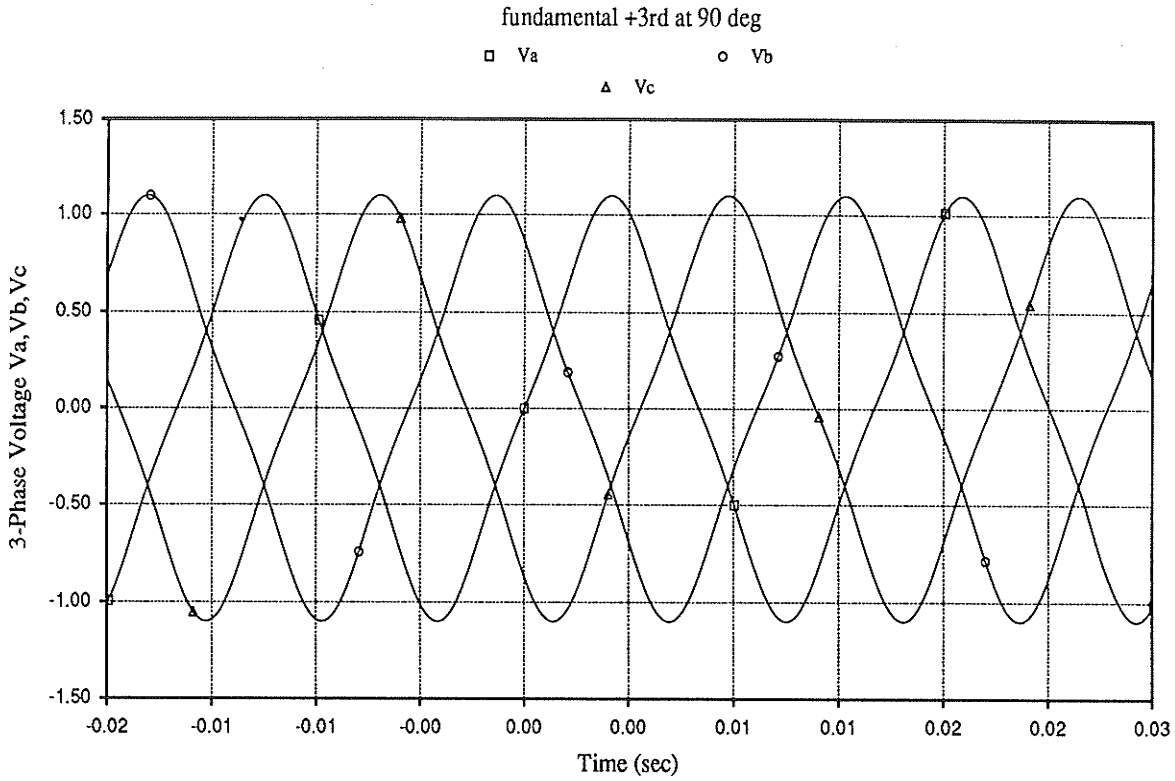


Figure 34. Voltage Signals with 10% 3rd Harmonic at γ=90°

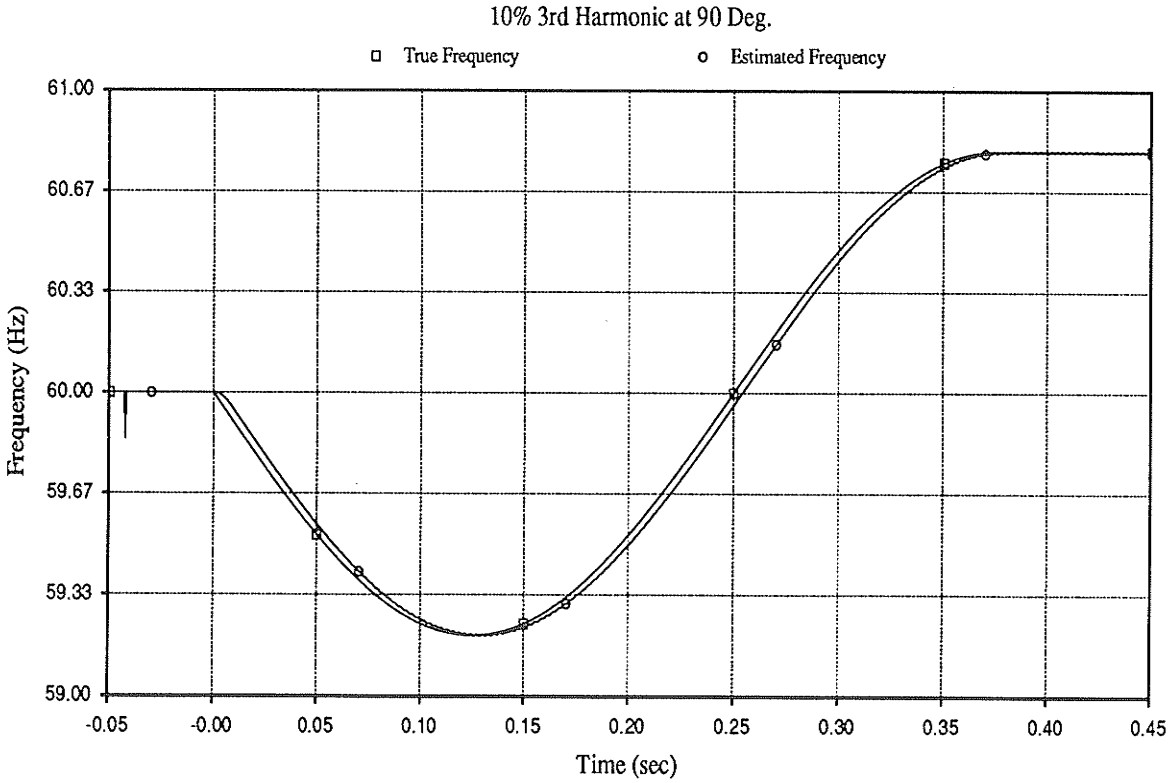


Figure 35. Result: 10% 3rd Harmonic at γ=90°

5.3. Frequency Swing Signal with 10% 3rd and 1% 5th Harmonics.

Figures (36.) and (37.) show the voltage signals and frequency response, respectively. Appendix (2) lists the program. The phase difference between the fundamental and the harmonics is kept at 0^{o}.

The estimated frequency in figure (37.) starts to ripple a bit. Fortunately, the oscillations are confined to a small range about the true frequency. The result from this test shows that the error becomes larger as higher harmonics are added. The algorithm is apparently sensitive to a higher order of harmonics, even with a small magnitude. The oscillations may be reduced and smoothed by averaging previous data without causing a long delay.

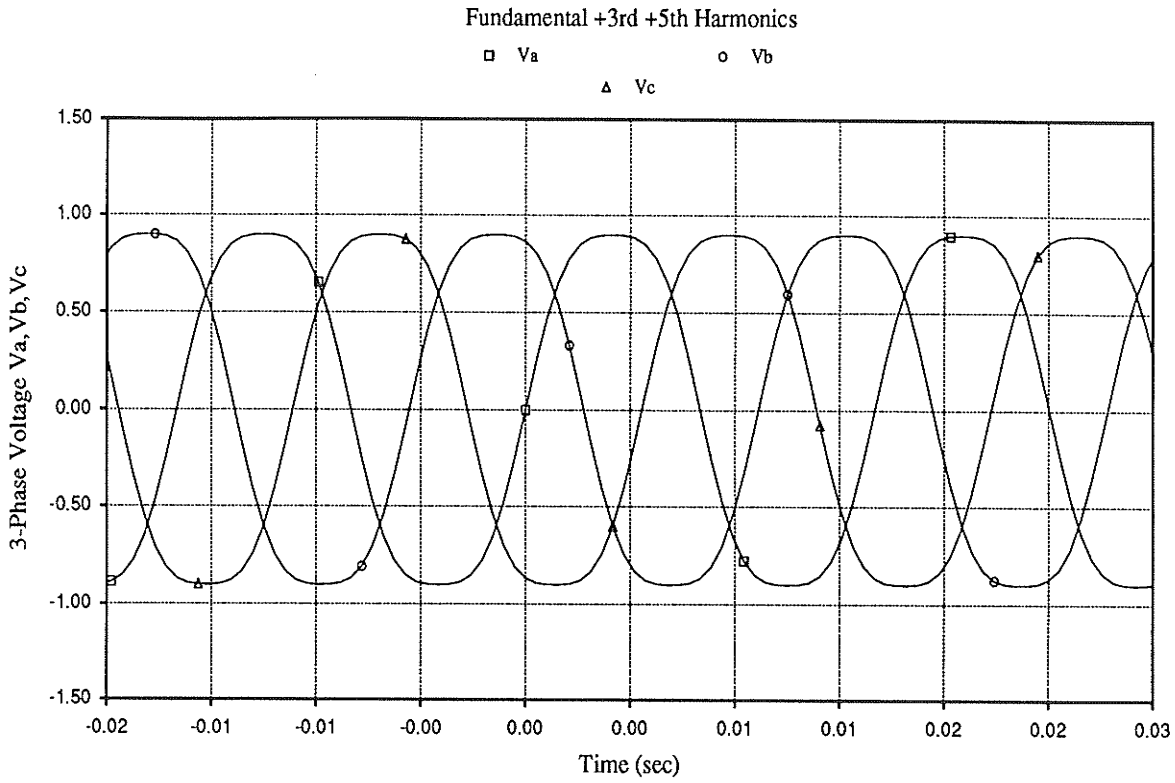


Figure 36. Voltage Signals with 10% 3rd and 1% 5th Harmonics at γ=0°

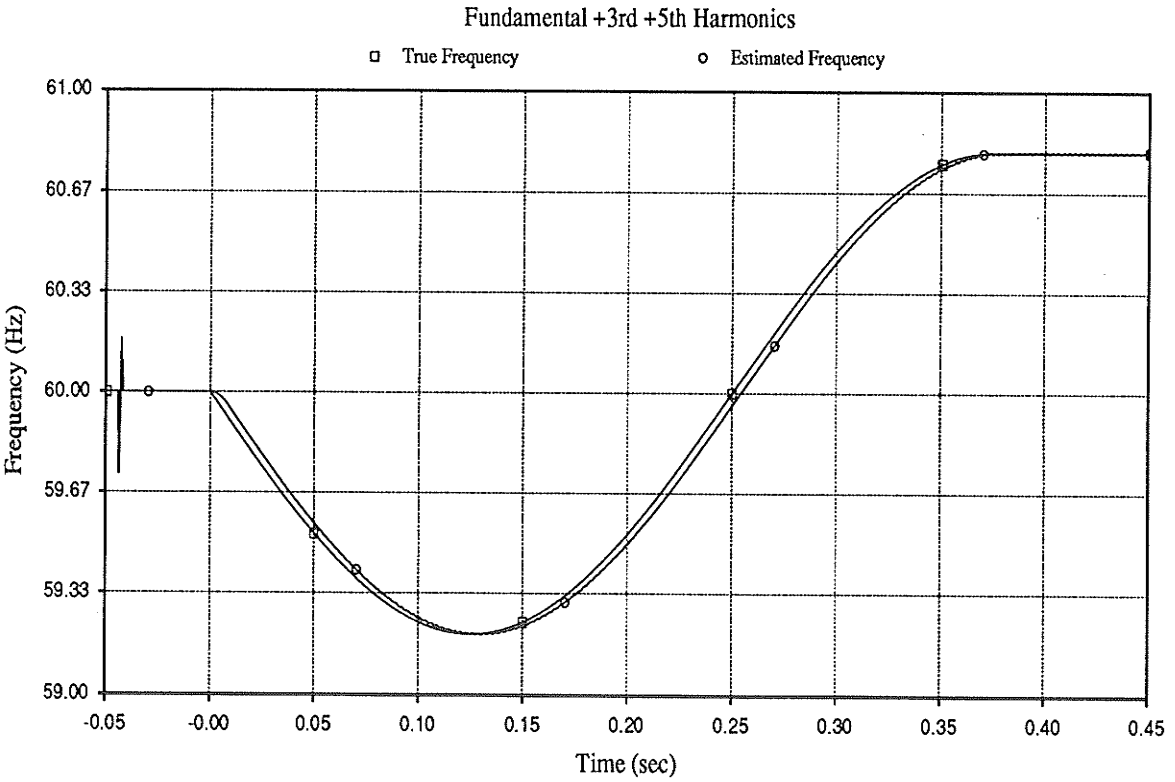


Figure 37. Result: 10% 3rd and 1% 5th Harmonics at γ=0°

5.4. Frequency Swing Signal with Gaussian Distributed High Frequency Random Noise

Figure (38.) shows the 3–phase voltage signals mixed with random noise, although it is hard to see because of the relatively small magnitude. Appendix (3) includes three computer programs: one to produce random noise, one to simulate Gaussian Distributed random noise and one frequency relay algorithm with a digital low–pass filter integrated.

As mentioned earlier, a quantization error noise is added to the digitized signals for this particular test. The results shown in figure (39.) support the conclusion drawn in section (5.3.) that as the high frequency component increases, the oscillation becomes larger. In order to solve this problem, a digital low–pass filter (NOT an anti–aliasing filter) is added to the program. The digital filter used extracts mainly the fundamental, and eliminates the odd order harmonics and attenuates high frequency components. After the input signals are fed through the digital filter, figure (40.) shows the resulting output. There is less oscillations in the frequency response. The output curve may be further improved by taking the average of, for example, 32 data points, and figure (41.) shows the result if it is done. There is significant time delay in frequency response after the filtering. The delay is about 8 milliseconds when compared to 4 milliseconds in the previous cases.

The testing of the algorithm under the above noise conditions is considered to be relatively harsh, as under normal circumstances the input signals are usually filtered with analog filters which are sometimes more effective.

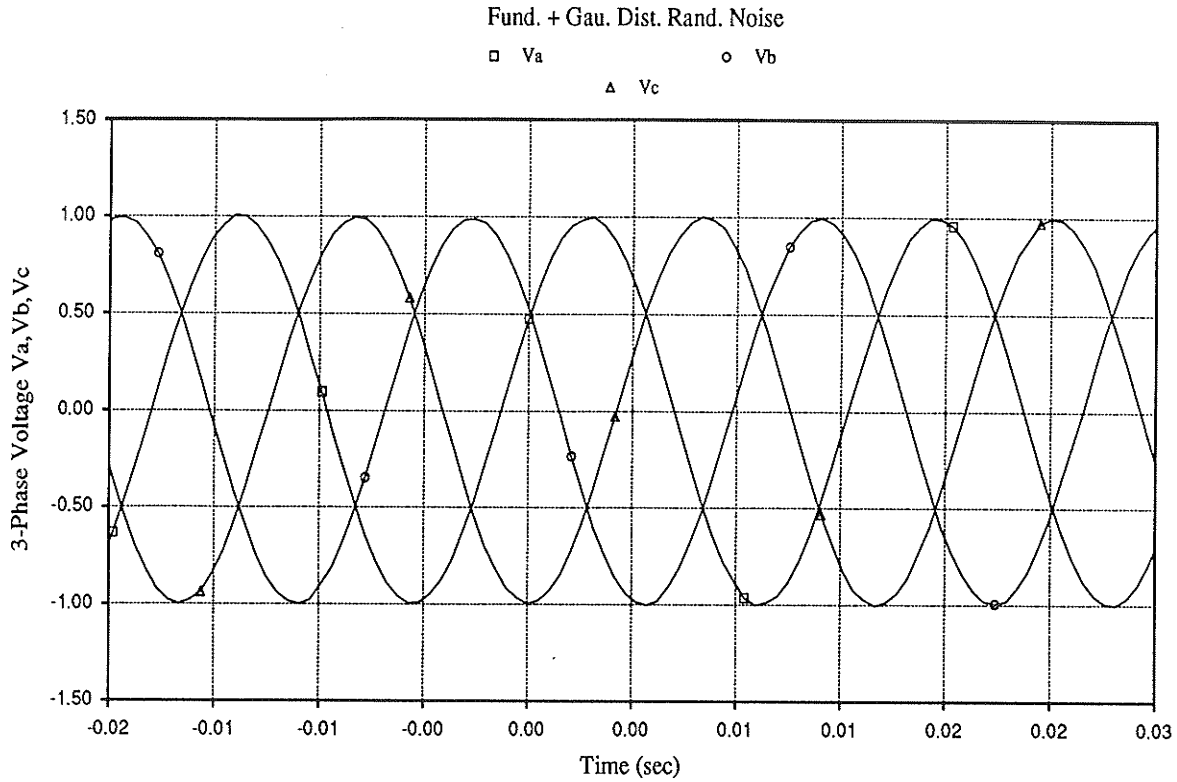


Figure 38. Voltage Signals with Gaussian Distributed High Frequency Random Noises

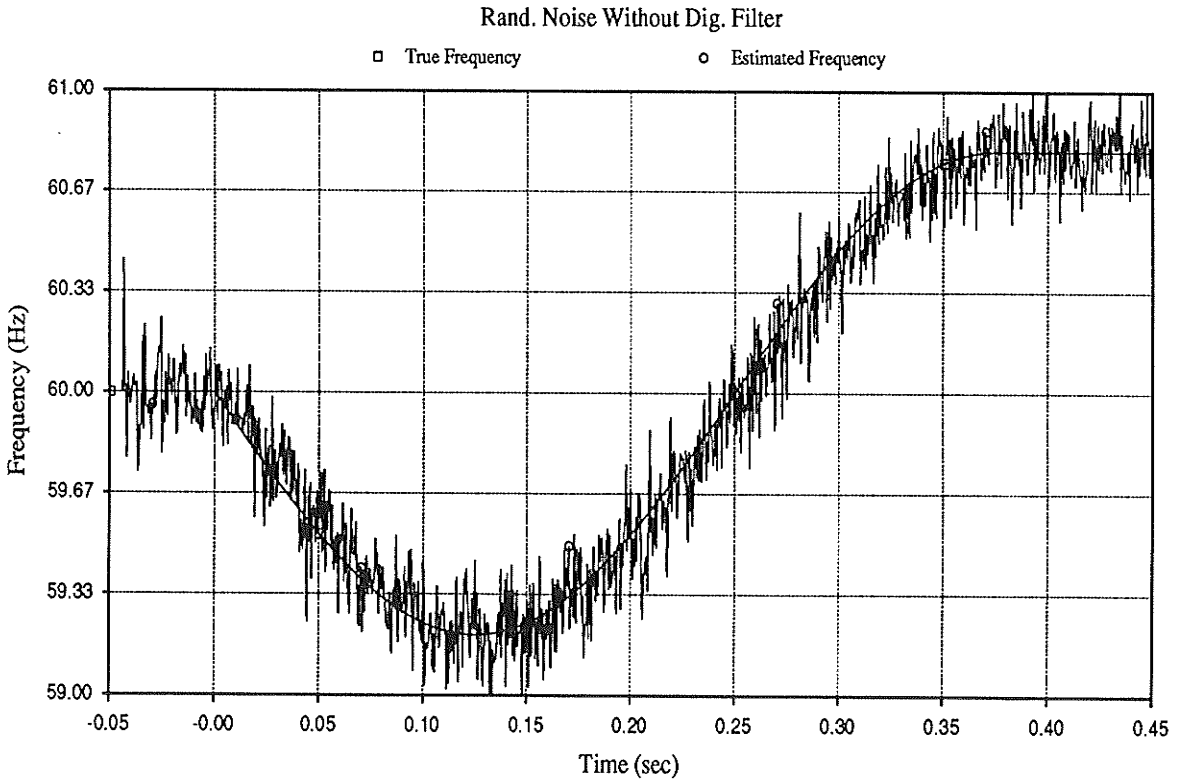


Figure 39. Result: Without Digital Low–Pass Filter: Gaussian Distributed High Frequency Random Noises

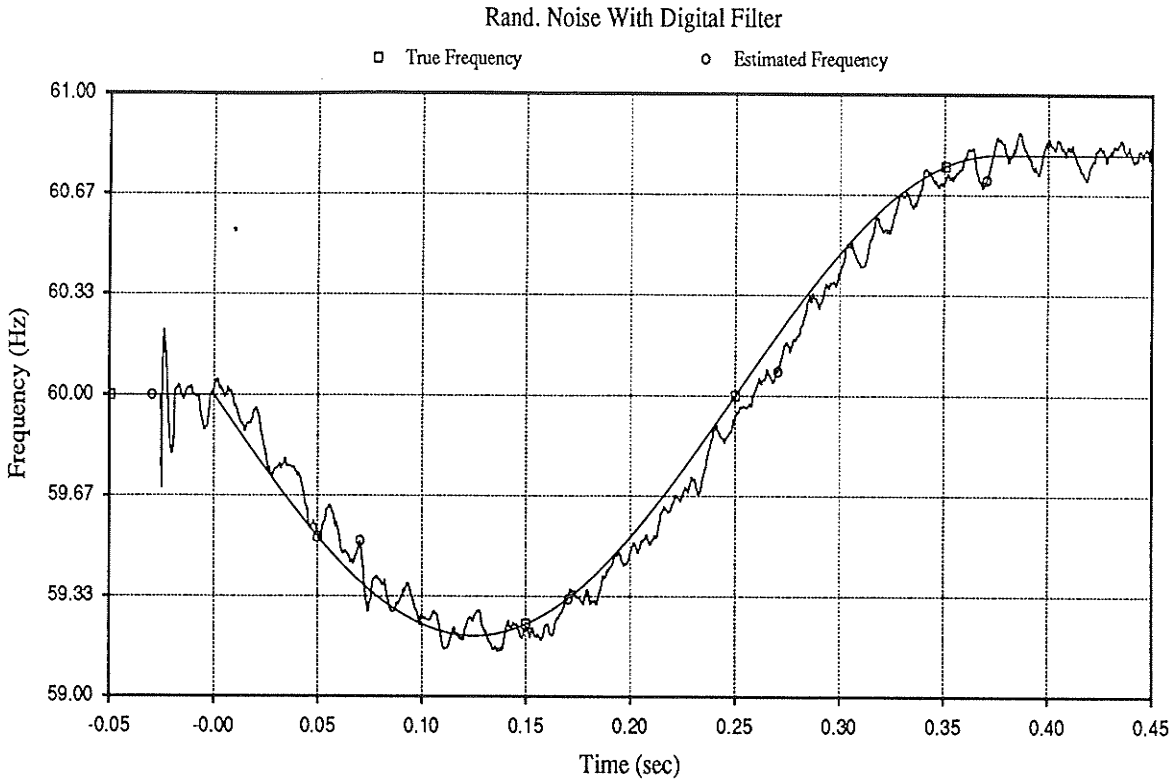


Figure 40. Result: With Digital Low–Pass Filter: Gaussian Distributed High Frequency Random Noises

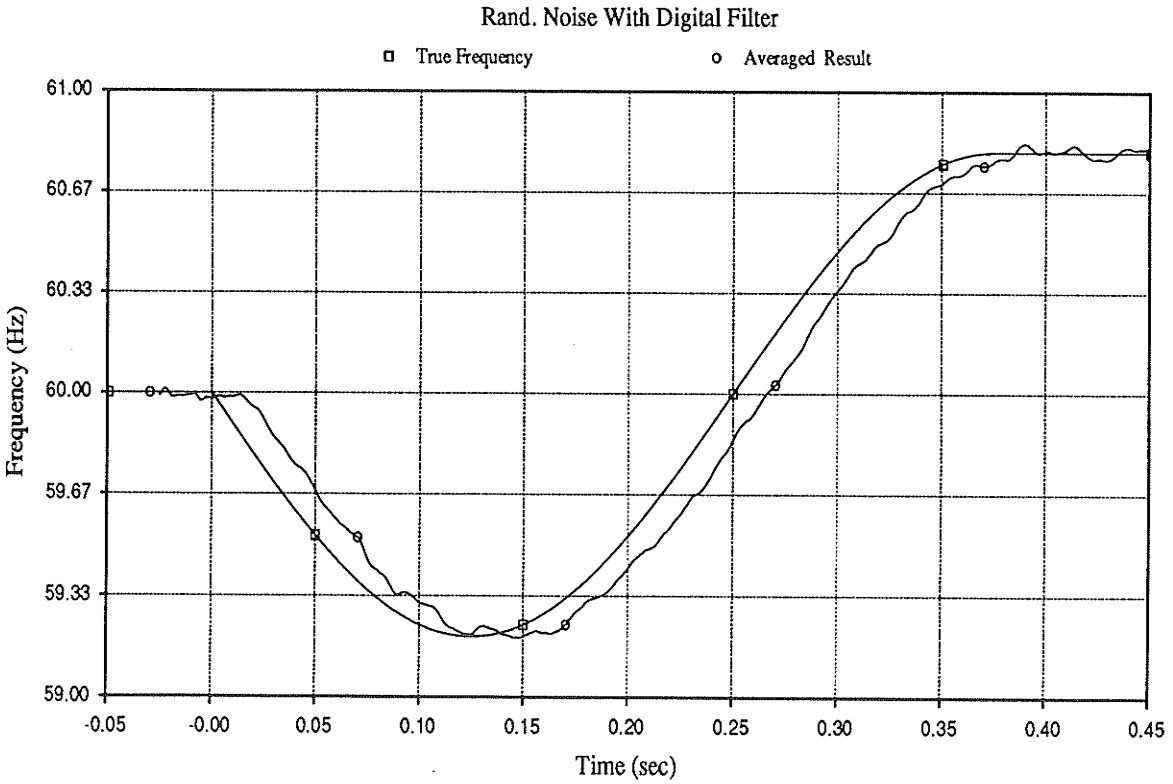


Figure 41. Result: Averaging the Output+ Digital Low Pass Filter

5.5. Fundamental Voltage Signal (60 Hz) with 50% Sudden Drop in Magnitude

Basically, there are not any frequency changes during the entire period of the test. There is only a sudden drop in voltage magnitude at time=0 seconds. The magnitude of the three–phase voltages after time=0 second is half of that before time=0 seconds, as shown in figure (42.). Two horizontal lines in figure (43.) overlap at 60 Hz indicating both the input and the output frequency are at a constant of 60 Hz. The algorithm reacts during the transition period, but the rate of frequency change limiter prevents a sudden frequency jump. Appendix (4) gives the list of the computer program.

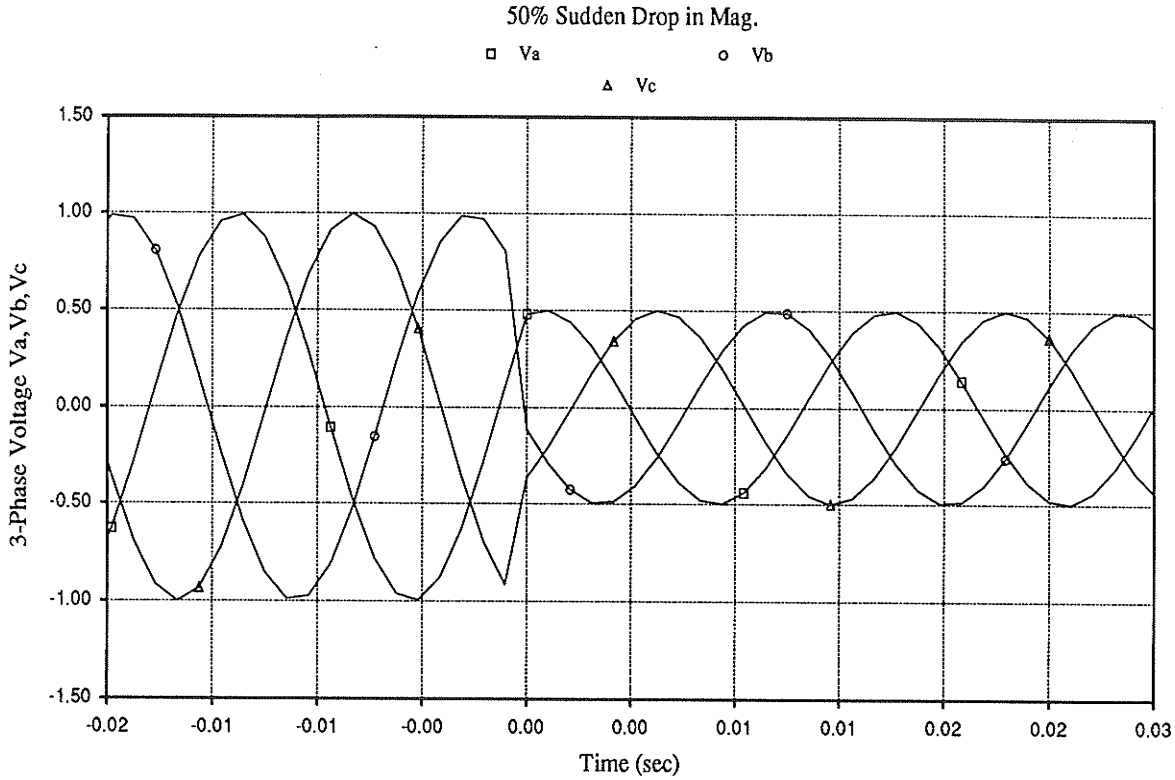


Figure 42. Fundamental Voltage Signals with 50% Sudden Drop in Magnitude at Time=0 seconds.

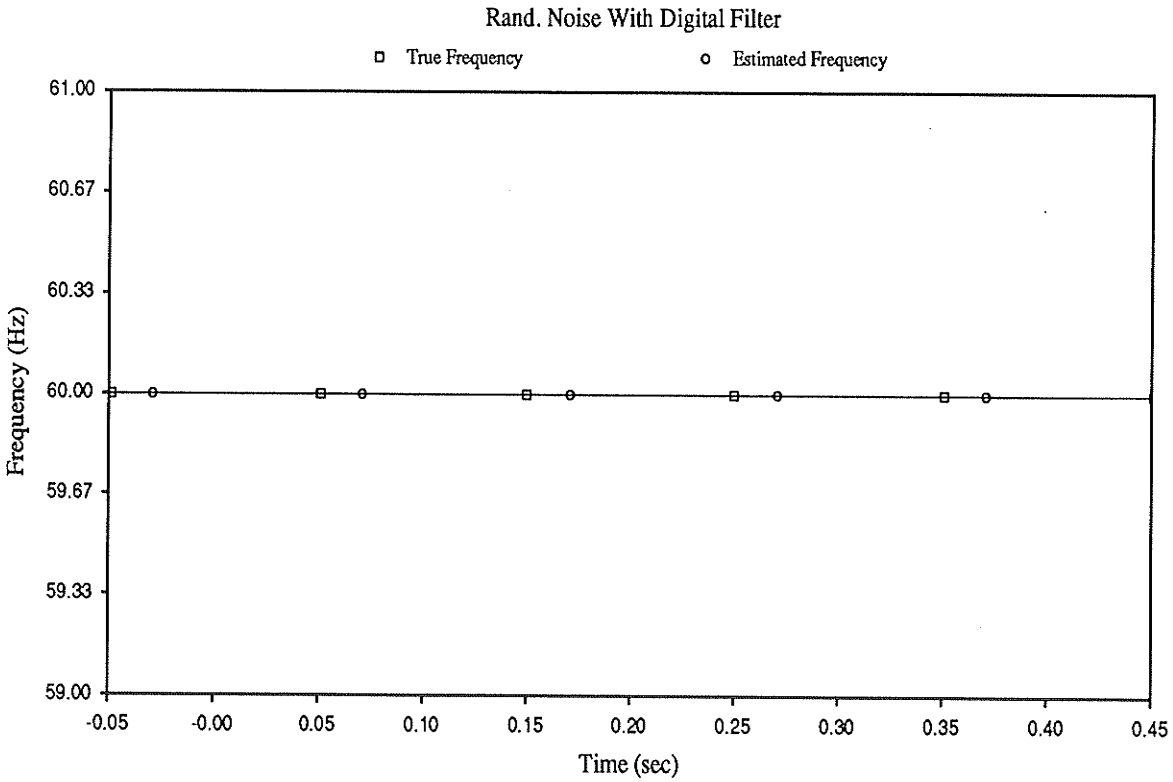


Figure 43. Result: 50% Sudden Drop in Magnitude

5.6. Fundamental Signal That Change Phase by –90 Degrees Over A Period of One Cycle

By looking at the voltage curves of figure (44.), the phase angles of the three signals shift by 90 degrees over a period of one cycle beginning at time=0 seconds. The frequency of the input signals during that period is only 45 Hz, and 60 Hz during rest of the time. Once again, the rate of frequency change limiter prevents the sudden jump at the two transition points. The computer program is listed in Appendix (5).

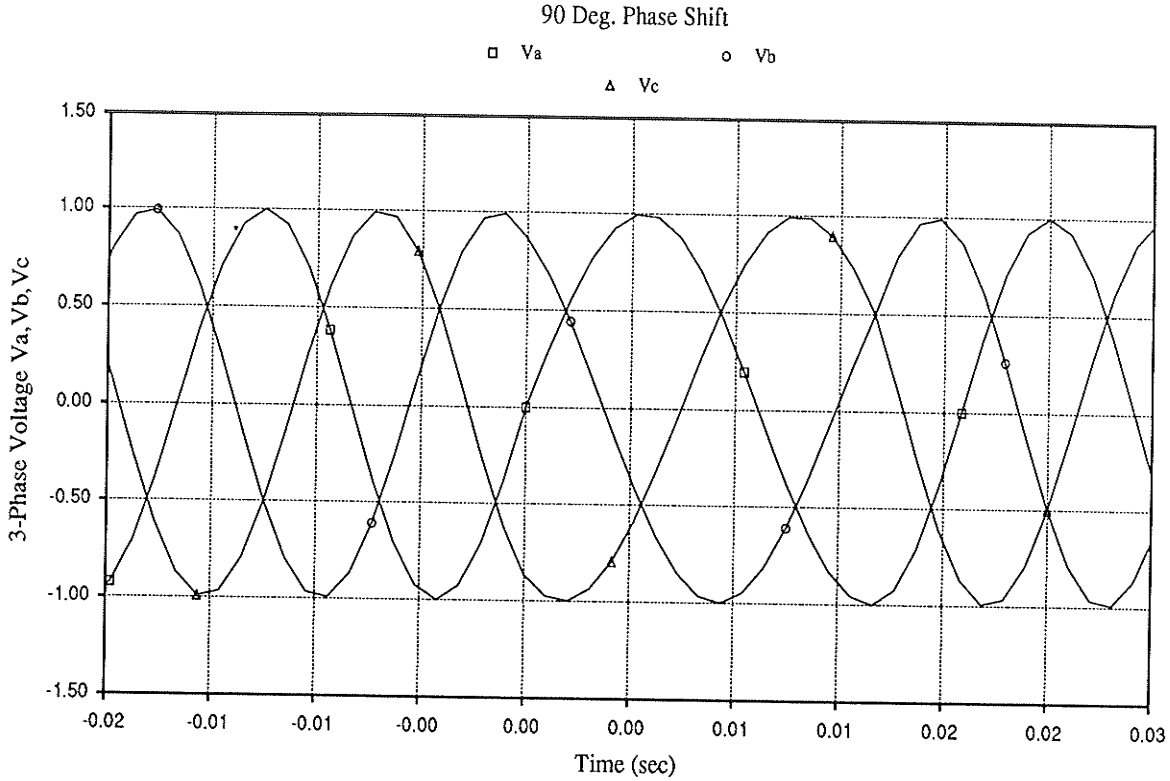


Figure 44. Fundamental Voltage Signals That Change Phase by –90 Degrees Over A Period of One Cycle.

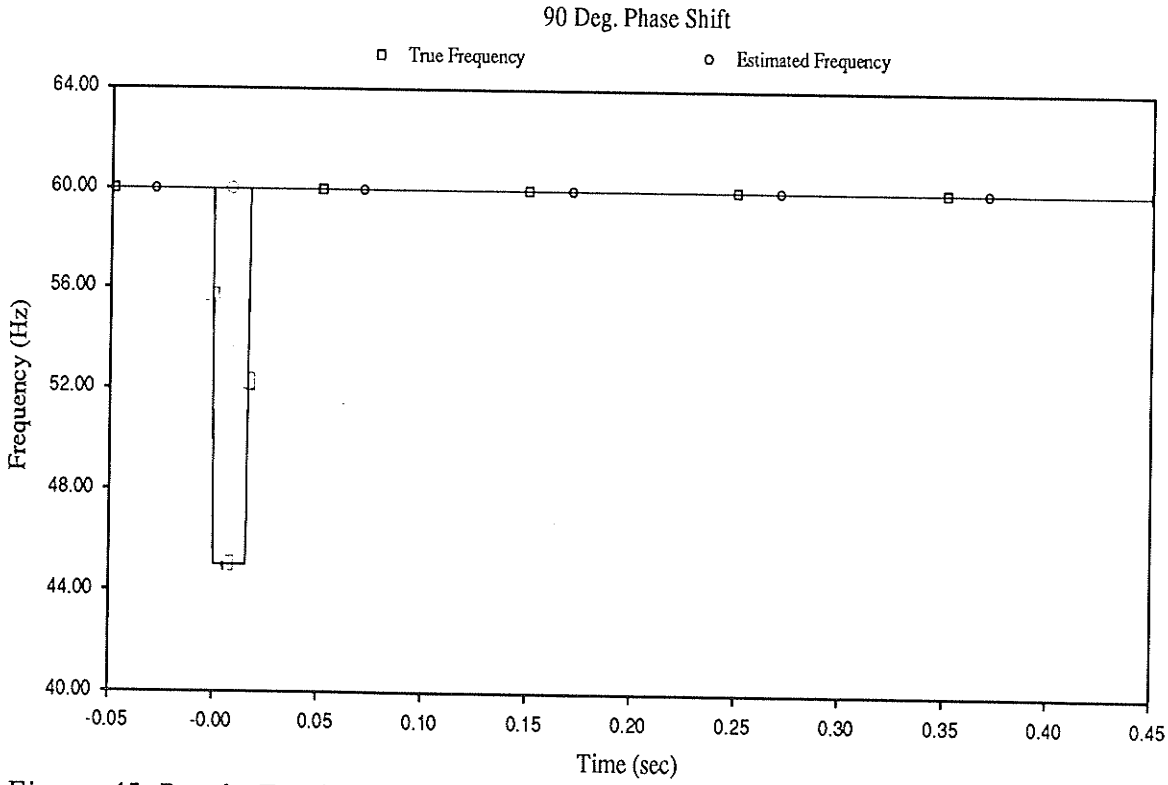


Figure 45. Result: Fundamental Signals That Change Phase by –90 Degrees

Chapter 6. Considerations on the Hardware and Software Implementation

6.1. Microprocessor Board Implementation Consideration

Despite the fact that we do not test our method on a microprocessor based device, the algorithm is relatively easy to implement with some modifications. However, since we have been using the floating point arithmetic (workstation processor), the error resulted from using the integer arithmetic microprocessor is expected to be larger. For a power protection scheme, a digital signal processor (DSP) is usually included in the device to facilitate the actual signal processing and floating–point based DSP is already available. Figure (46.) shows the modified algorithm for the implementation. The digitized input signals are fed into the DSP which perform a DFT and returns the fundamental component to the central processing unit (CPU). Since it is difficult to perform trigonometrical calculations on a microprocessor and it is very time consuming, an alternative is to look up the inverse tangent values from a stored table.

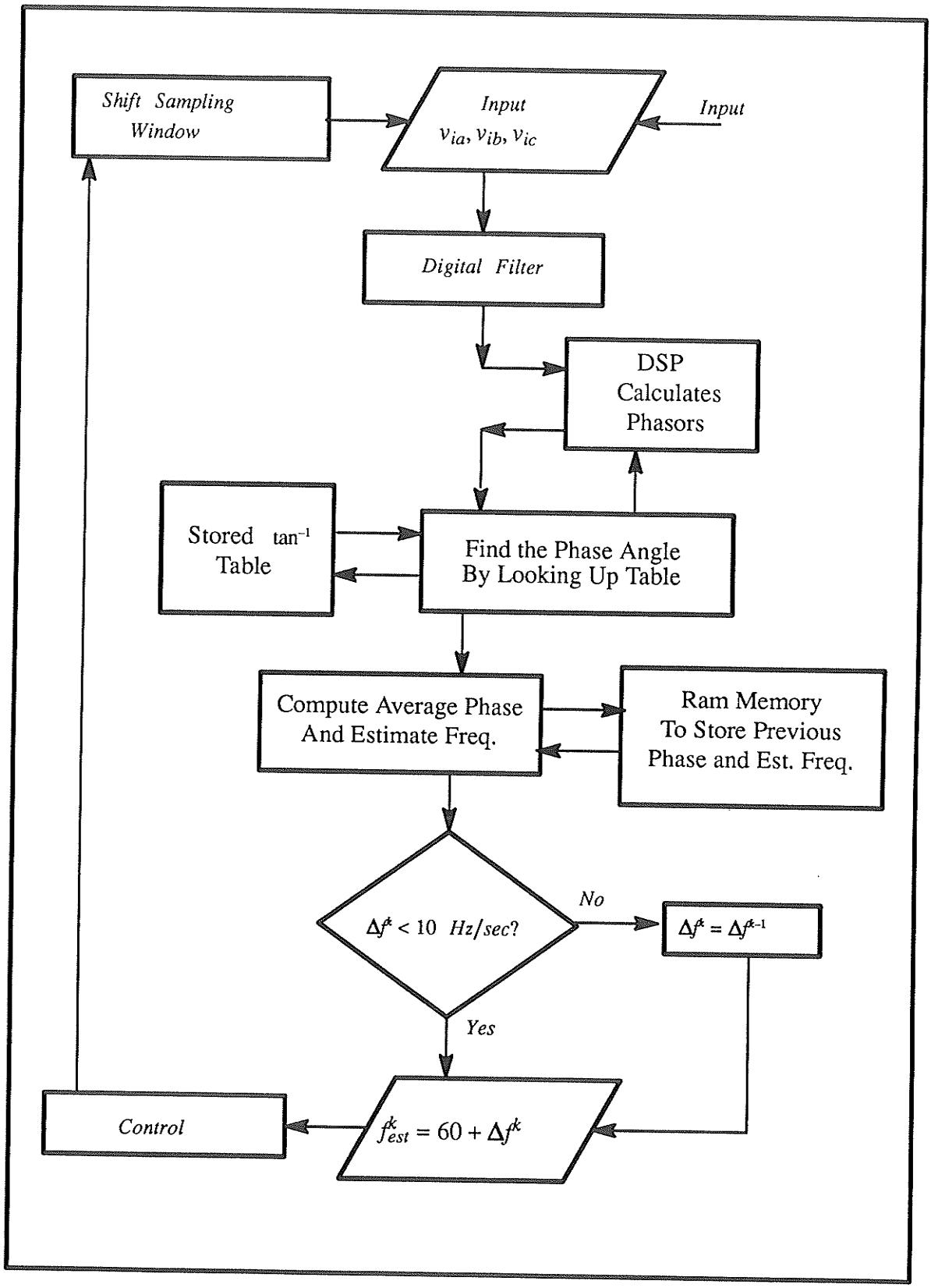


Figure 46. Modified Algorithm for Microprocessor Implementation

6.2. Testing the Algorithm with A Larger Frequency Swing Range.

Although our algorithm is based on a fixed sampling rate and a small deviation of frequency, the algorithm will be tested for signals with a frequency swing of 20 Hz. The aim is to resemble a more practical situation in which a protection relay must accommodate power system changes. Figure (47.) shows that the result is still satisfactory under the above conditions. The frequency varies from 60 to 50 Hz, then to 70 Hz. The scale is enlarged in Figure (48.) to show the delay time.

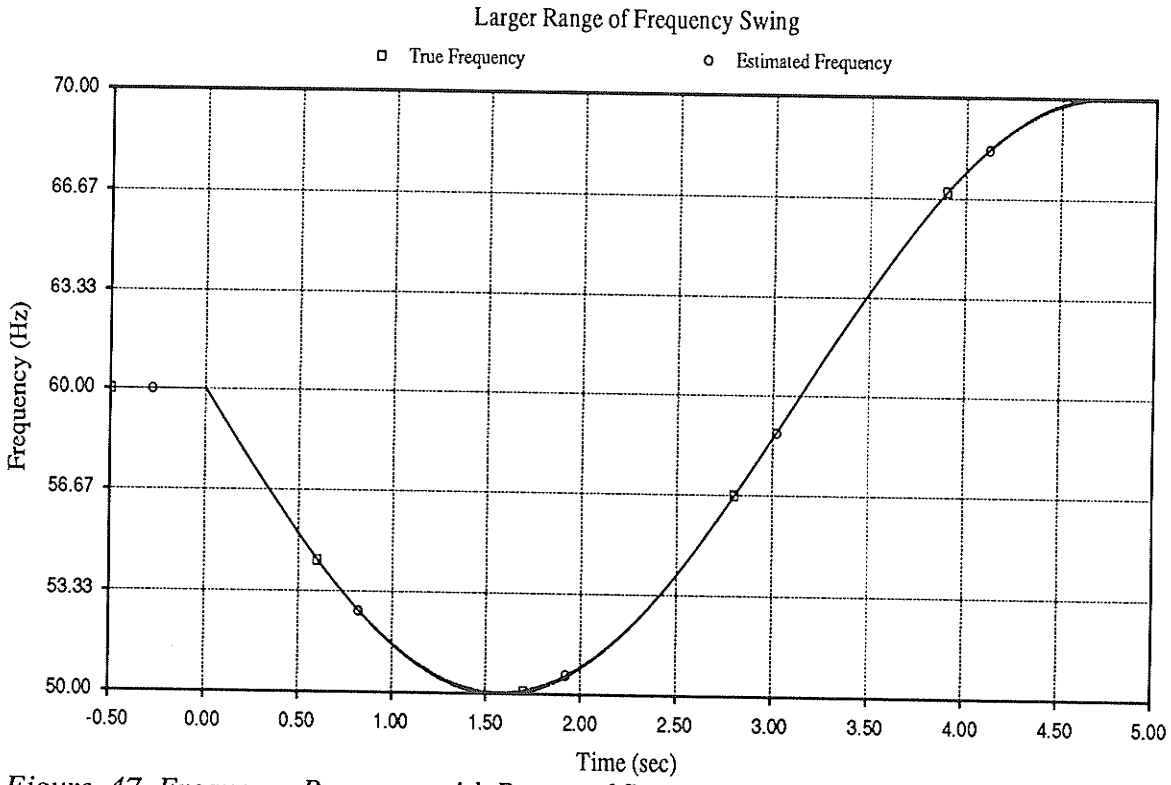


Figure 47. Frequency Response with Range of Swing of 20 Hz.

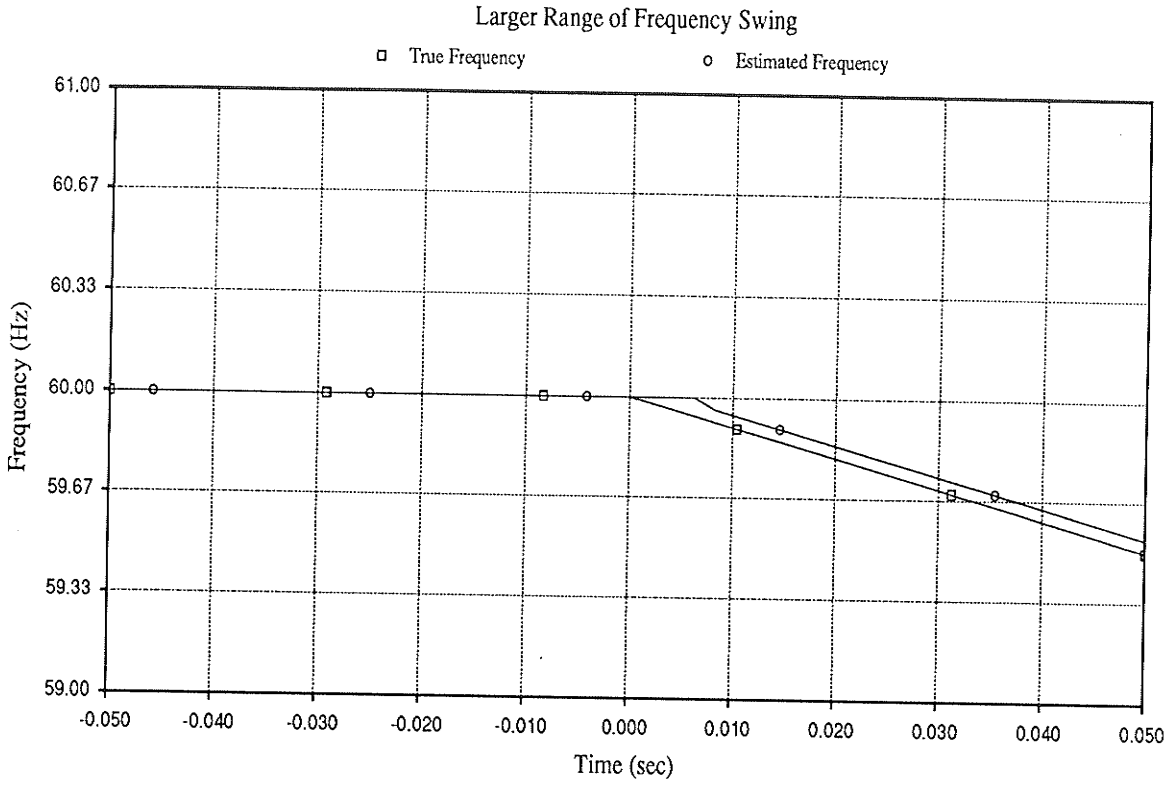


Figure 48. Frequency Response with Range of Swing of 20 Hz (Scale Enlarged)

6.3. Adaptive Relaying

"Adaptive protection is a protection philosophy which permits and seeks to make adjustment to various protection functions in order to make them more attuned to prevailing power system conditions" [12]. In other words, an ideal protecting relay should be able to adapt to the ever changing power system conditions. Relays with fixed settings are insufficient because not all conceivable network conditions can be anticipated. For a frequency relay based on the Fourier Transform, the sampling rate is recommended to be updated after each computation. The algorithm has been tested with modified feedback sampling rate and 4 samples per cycle. Figure (49.) and figure (50.) shows the results with feedback sampling rate. The delay time is reduced when compared to the result without the feedback.

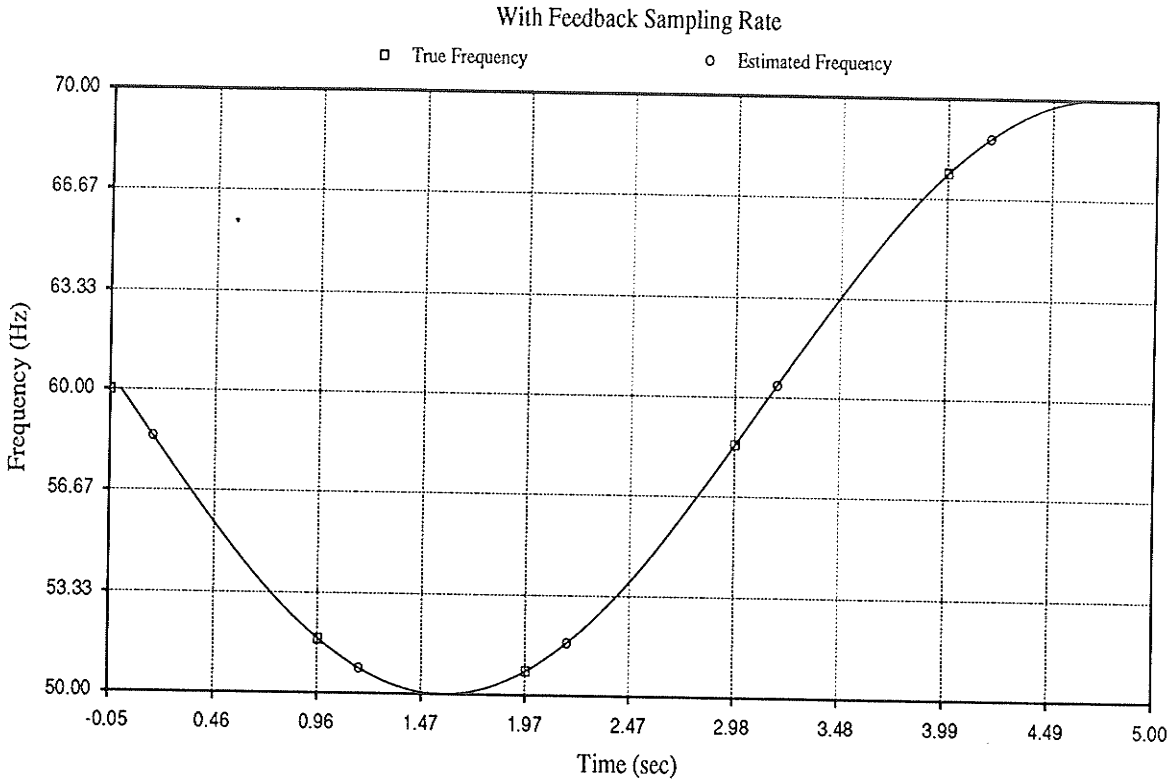


Figure 49. Frequency Response Feedback Sampling Rate.

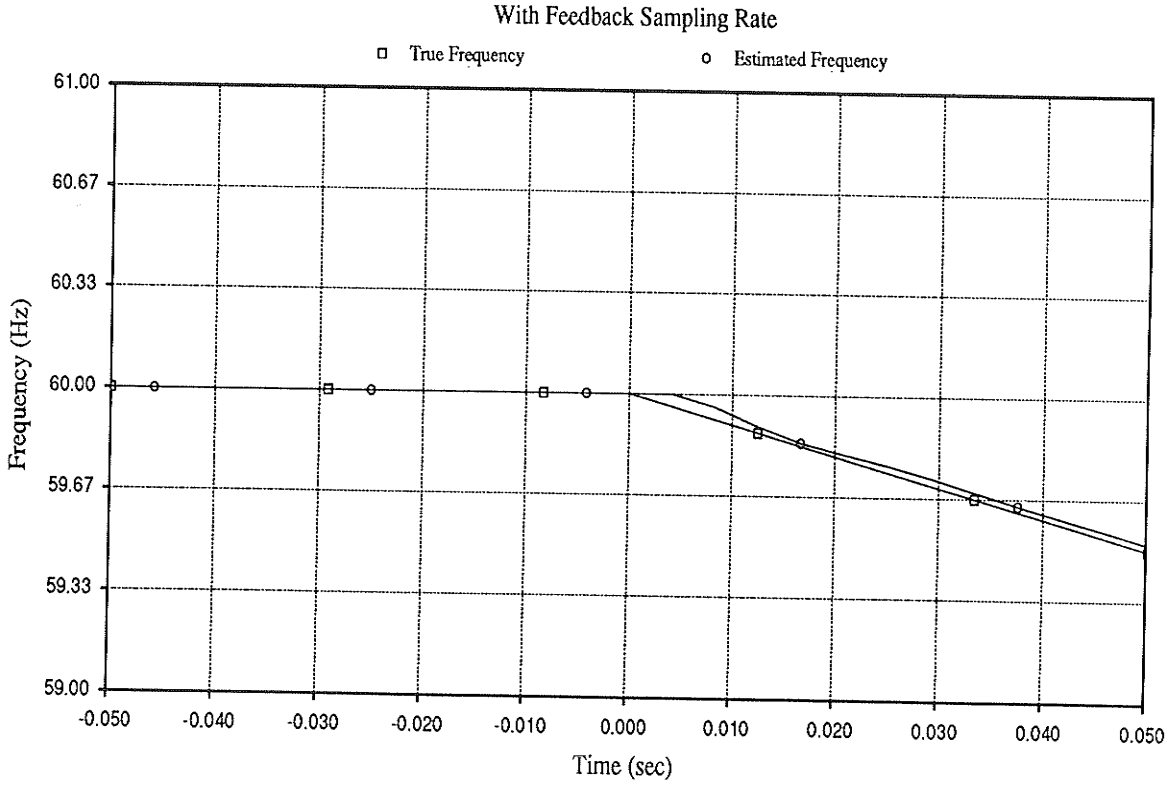


Figure 50. Frequency Response Feedback Sampling Rate (Scale Enlarged).

Chapter 7. Conclusions

1. The Fourier Series is one of the most commonly used algorithms in computer relaying. The Discrete Fourier Transform (DFT) provide useful information to frequency relays by extracting the fundamental frequency component of incoming signals.

2. The new and modified DFT algorithm uses only half of the window data points in each calculation, thus reducing the delay.

3. Using the average of three single phase calculation further reduces the delay.

4. A varying frequency test signal must be defined by a corresponding varying phase function.

5. The new algorithm works very well with the following test signals, which simulate a typical situation for a power system:

   i.) Pure frequency swing voltage signal.
   ii.) Frequency swing signal with 10% 3rd harmonic at various overlapping angles.
   iii.) Frequency swing signal with 10% 3rd and 1% 5th harmonics.
   iv.) Frequency swing signal with Gaussian distributed high frequency random noises.
   v.) Fundamental signal with 50% sudden drop in magnitude.
   vi.) Fundamental signal that changes phase by –90 degrees over a period of one cycle.

Consequently, the algorithm is able to track a signal with constant frequency as well as a swinging frequency, provided that the signal is anti–aliased (the algorithm is very sensitive to the noise) and its rate of change of frequency is less than 10 Hz/sec.

References

[1] W.D. Stevenson, Jr., *Element of Power System Analysis*, McGraw–Hill, New York 4th Edition, 1982.

[2] M. A. Laughton and M.G. Say, *Electrical Engineers' Reference Book*, Butterworth & Co (Publisher) Ltd, Chapter 15, 1985.

[3] A.R. van C., Warrington, *Protection Relay*, Vol. 1, Chapman & Hall, 1962.

[4] G.K. Laycock, "*Digital Signal Processing Techniques Applied to Power System Protection*", Ph.D thesis, July 1972.

[5] J.G. Gilbert, E.A. Udren and M. Sackin, "*Evaluation of Algorithm for Computer Relaying*", IEEE PAS summer meeting, Mexico City, Mexico, 1977.

[6] L.W. Couch II, *Digital and Analog Communication Systems*, MacMillan, 2nd edition, New York, 1987.

[7] A.V. Oppenheim, A.S. Willsky and I.T. Young, *Signals and Systems*, Prentice–Hill, New Jersey, 1983.

[8] A.G. Phadke, J.S. Thorp and M.G. Adamiak, "*A New Measurement Technique for Tracking Voltage Phasors, Local System Frequencies and Rate of Change of Frequency*", IEEE Transaction on Power Apparatus and Systems, Vol.PAS–102, No. 5, pp 1025–1038, May 1983.

[9] G.W. Swift, Class Notes.

[10] W.H. Press, B.P Flanneyery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge, England, 1989.

[11] S.M. Bozic, *Digital and Kalman Filtering*, Macmillan, India, 1979.

[12] A.G. Phadke and J.S. Thorp, *Relaying for Power Systems*, Research Studies Press Ltd., Somerset, England, 1988.

Additional Sources of Information

G.B. Benmouyal, "*An Adaptive Sampling–Interval Generator for Digital Relaying*", IEEE Transactions on Power Delivery, Vol. 4. No. 3, pp 1603–1609, July 1989.

H. Tao and I.F. Morrison, "*The Measurement of Power System Frequency Using a Microprocessor*", Electric Power System Research, Vol. 11, pp 103–108, 1986.

A. A. Girgis and F.M. Ham, "*A New FFT–Based Digital Frequency Relay for Load Shedding*", IEEE Transactions on Power Apparatus and Systems, Vol. PAS–101, No.2, pp 433–439, February 1982.

M.S. Sachdev and M.M. Giray, "*Measurement of Local Frequency from Digitized Bus Voltage Samples*", Power System Research Group Paper, University of Saskatchewan, Saskatoon, Saskatchewan.

M.S Sachdev and M.M. Giray, "*A Digital Frequency and Rate of Change of Frequency Relay*", Trans. CEA. Engineering and Operation Division, Vol. 17, Part 3, 1978, No. 78–Sp–145.

M.S Sachdev and M.M. Giray, "*Amplitude and Frequency Measurements by Discrete Fourier Transform and Least Error Squares Techniques*", Trans. CEA, Engineering and Operation Division, Spring Meeting, Montreal, 1985.

Microprocessor Relays and Protection Systems, Tutorial Course, A Continuing Education Service of the IEEE Power Engineering Society. 1988.

Appendix 1

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program estimates the signal frequency using modified       #
#  Discrete Fourier Transform technique.                                     #
#                                                                            #
#  Input:  simulated 3-phases voltages                                       #
#  Output: estimated frequency                                               #
#                                                                            #
#  Signal component: Pure Frequency Swing Signal                             #
#                                                                            #
##############################################################################
*/


#include <stdio.h>                          /* header */
#include <math.h>


#define ND              32      /* no. of samples taken per cycle */
#define ND_OVER_2       ND/2            /* half window calculation       */
#define PI              3.141592
#define  MAG            1.0             /* magnitude of the voltage */

#define PHI_A           0.5             /* arbitary angles chosen so that */
#define PHI_B           0.1             /* curves are continuous at       */
#define PHI_C           -1.785          /* transition */

#define THETA_A         0.0                     /* phase angle diff. */
#define THETA_B         THETA_A+2.0*PI/3        /* for 3 phases       */
#define THETA_C         THETA_A+4.0*PI/3

#define TIMELIMIT       0.5             /*  simulation time   */
#define  MAXDF          10.0/ND/60.0+0.01 /* 10 Hz/sec max.change */

/*
##############################################################################
#                                                                            #
#  Purpose: This function calculate  the true frequency of the input         #
#           signal                                                           #
#                                                                            #
#  Input:  time in seconds                                                   #
#  Output: true frequency                                                    #
#                                                                            #
#  true frequenc = 60.0 Hz              for t < 0                            #
#                = time varying             0<= t  <=0.375 sec.            #
#                =60.8 Hz                   0.375 < t                      #
#                                                                            #
##############################################################################
*/

double Func_true_freq(t)
double t;
{
double true_freq;
```

```
        if(t<0.0)
        true_freq=60.0;
        else if ((t>=0.0)&&(t<0.375))
        true_freq=60.0-0.8*sin(4*PI*t);
        else
        true_freq=60.8;

return(true_freq);
}

/*
##############################################################################
#                                                                            #
#  Purpose: These 3 functions simulates the input 3-phases voltages          #
#                                                                            #
#  Input: time in seconds                                                    #
#  Output: 3-phase voltage signal                                            #
#                                                                            #
##############################################################################

*/

double In_signal_a(t)
double t;
{
double signal;

 if(t<0.0)
        signal=sin(2.0*PI*60.0*t+PHI_A);
        else if ((t>=0.0)&&(t<0.375))
        signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B);
        else
        signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C);

return(signal);
}



double In_signal_b(t)
double t;
{
double signal;

        if(t<0.0)
        signal=sin(2.0*PI*60.0*t+PHI_A+THETA_B);
        else if ((t>=0.0)&&(t<0.375))
        signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B+THETA_B);
        else
        signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C+THETA_B);

return(signal);
}


double In_signal_c(t)
```

```
double t;
{
double signal;

        if(t<0.0)
        signal=sin(2.0*PI*60.0*t+PHI_A+THETA_C);
        else if ((t>=0.0)&&(t<0.375))
        signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B+THETA_C);
        else
        signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C+THETA_C);


return(signal);
}



/*
###############################################################################
#                                                                             #
#   Main Program Starts                                                       #
#                                                                             #
###############################################################################

 */

main()
{

int     n,M,count;
double  Buffer_Va[ND_OVER_2],Buffer_Vb[ND_OVER_2],Buffer_Vc[ND_OVER_2];
double  Buffer_Va_cos[ND_OVER_2],Buffer_Va_sin[ND_OVER_2];
double  Buffer_Vb_cos[ND_OVER_2],Buffer_Vb_sin[ND_OVER_2];
double  Buffer_Vc_cos[ND_OVER_2],Buffer_Vc_sin[ND_OVER_2];
double  Func_true_freq(),In_signal_a(),In_signal_b(),In_signal_c();
double  t,freq_est_last,freq_est_current,freq_true_last,freq_true_current;
double  Va,Vb,Vc,max_df;
double  sum_Va_cos,sum_Va_sin,sum_Vb_cos,sum_Vb_sin,sum_Vc_cos,sum_Vc_sin;
double  angle_a, angle_b,angle_c,angle_ave_last,angle_ave_current;
double  df_dt_true,df_dt_est;

/*
###############################################################################
#                                                                             #
#   Initilization of variables                                                #
#                                                                             #
###############################################################################
*/

M=0;
t=-0.05;
freq_est_last=60.0;
freq_true_last=60.0;
angle_ave_last=0.0;

count=4;
```

```
printf("hello\n");

/*
###############################################################################
#                                                                             #
#   Initilization of Buffers                                                  #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
Buffer_Va_cos[n]=0.0;
Buffer_Va_sin[n]=0.0;
Buffer_Vb_cos[n]=0.0;
Buffer_Vb_sin[n]=0.0;
Buffer_Vc_cos[n]=0.0;
Buffer_Vc_sin[n]=0.0;
}


/*
###############################################################################
#                                                                             #
#  While Loop                                                                 #
#                                                                             #
###############################################################################
*/


while (t<TIMELIMIT)
{



freq_true_current=Func_true_freq(t);


/*
###############################################################################
#                                                                             #
#  Get 3-phase voltages by calling functions                                  #
#                                                                             #
###############################################################################
*/

Va=In_signal_a(t);
Vb=In_signal_b(t);
Vc=In_signal_c(t);


/*
```

```
/*
###############################################################################
#                                                                             #
#  Shift the correlation product buffers (or window) once                     #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2-1;n++)
{

Buffer_Va_cos[n]=Buffer_Va_cos[n+1];
Buffer_Va_sin[n]=Buffer_Va_sin[n+1];
Buffer_Vb_cos[n]=Buffer_Vb_cos[n+1];
Buffer_Vb_sin[n]=Buffer_Vb_sin[n+1];
Buffer_Vc_cos[n]=Buffer_Vc_cos[n+1];
Buffer_Vc_sin[n]=Buffer_Vc_sin[n+1];
}


if (M==ND) M=0;

/*
###############################################################################
#                                                                             #
#  Calculate the correlation product for the most recent input signals        #
#  by multiplying voltages and the reference cosine and sine curves           #


#                                                                             #
###############################################################################
*/

Buffer_Va_cos[ND_OVER_2-1]=Va*cos(2*PI*(M%ND)/ND);
Buffer_Va_sin[ND_OVER_2-1]=Va*sin(2*PI*(M%ND)/ND);
Buffer_Vb_cos[ND_OVER_2-1]=Vb*cos(2*PI*(M%ND)/ND);
Buffer_Vb_sin[ND_OVER_2-1]=Vb*sin(2*PI*(M%ND)/ND);
Buffer_Vc_cos[ND_OVER_2-1]=Vc*cos(2*PI*(M%ND)/ND);
Buffer_Vc_sin[ND_OVER_2-1]=Vc*sin(2*PI*(M%ND)/ND);

M+=1;


sum_Va_cos=0.0;
sum_Va_sin=0.0;
sum_Vb_cos=0.0;
sum_Vb_sin=0.0;
sum_Vc_cos=0.0;
sum_Vc_sin=0.0;


/*
###############################################################################
#                                                                             #
#  Summation of all the correlation products                                  #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
```

```
{
sum_Va_cos+=Buffer_Va_cos[n];
sum_Va_sin+=Buffer_Va_sin[n];
sum_Vb_cos+=Buffer_Vb_cos[n];
sum_Vb_sin+=Buffer_Vb_sin[n];
sum_Vc_cos+=Buffer_Vc_cos[n];
sum_Vc_sin+=Buffer_Vc_sin[n];
}


/*
###############################################################################
#                                                                             #
# Calculate the phase angles of the 3 phasors                                 #
#                                                                             #
###############################################################################
*/

angle_a=atan(sum_Va_cos/sum_Va_sin);
angle_b=atan(sum_Vb_cos/sum_Vb_sin);
angle_c=atan(sum_Vc_cos/sum_Vc_sin);

/*
###############################################################################
#                                                                             #
# Find the average of 3 phasores' angles                                      #
#                                                                             #
###############################################################################
*/

angle_ave_current=(angle_a+angle_b+angle_c)/3;


/*
###############################################################################
#                                                                             #
# Estimate the frequency of input signal                                      #
#                                                                             #
###############################################################################
*/

freq_est_current=60.0+(angle_ave_current-angle_ave_last)*60.0*ND/2.0/PI;

/*
###############################################################################
#                                                                             #
# df/dt limitor                                                               #
#                                                                             #
###############################################################################
*/
if(fabs((freq_est_current-freq_est_last))>MAXDF)
freq_est_current=freq_est_last;


/*
###############################################################################
```

```
#                                                                              #
# Calculate the df/dt                                                          #
#                                                                              #
################################################################################
*/

df_dt_true=(freq_true_current-freq_true_last)*60.0*ND;
df_dt_est=(freq_est_current-freq_est_last)*60.0*ND;


if(count==4)

{

printf(" %f  %f %f %f %f %f %f
 %f\n",t,freq_true_current,freq_est_current,Va,Vb,Vc,df_dt_true,df_dt_est);
count=0;
} /*if*/

/* df_dt_true,df_dt_est, */

freq_true_last=freq_true_current;
freq_est_last=freq_est_current;
angle_ave_last=angle_ave_current;

t+=1.0/ND/60.0;

count+=1;
} /*while*/

}
```

Appendix 2

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program estimates the signal frequency using modified       #
#  Discrete Fourier Transform technique.                                     #
#                                                                            #
#  Input:  simulated 3-phases voltages                                       #
#  Output: estimated frequency                                               #
#                                                                            #
#  Signal component:  Pure Frequency Swing Signal+                           #
#                      3rd (+5th) harmonics                                  #
#                                                                            #
##############################################################################
*/

#include <stdio.h>                          /* header */
#include <math.h>


#define ND              32                  /* no. of samples taken per cycle */
#define ND_OVER_2       ND/2                /* half window calculation */
#define PI              3.141592
#define  MAG            1.0                 /* magnitude of the voltage */

#define THETA_A         0.0                         /* phase angle diff. */
#define THETA_B         THETA_A+2.0*PI/3            /* for 3 phases       */
#define THETA_C         THETA_A+4.0*PI/3

#define TIMELIMIT       0.5
#define  MAXDF          10.0/ND/60.0+0.01 /* max freq change */
#define THIRD_H         10.0                /* % of third harmonics */
#define FIFTH_H         0.0                 /* % of fifth harmonics  1% max */
#define PHASE_SHIFT     PI                  /* phase diff. b/w fundamental */
                                    /* and the 3rd harmonics */


/*
##############################################################################
#                                                                            #
#  Purpose: This function calculate  the true frequency of the input         #
#           signal                                                           #
#                                                                            #
#  Input:  time in seconds                                                   #
#  Output: true frequency                                                    #
#                                                                            #
#  true frequenc = 60.0 Hz                for t < 0                          #
#                = time varying               0<= t  <=0.375 sec.           #
#                =60.8 Hz                     0.375 < t                     #
#                                                                            #
##############################################################################
*/

double Func_true_freq(t)
double t;
{
double true_freq;
```

```
 if(t<0.0)
 true_freq=60.0;
 else if ((t>=0.0)&&(t<0.375))
 true_freq=60.0-0.8*sin(4*PI*t);
 else
 true_freq=60.8;

return(true_freq);
}


/*
##############################################################################
#                                                                            #
#  Purpose: These 3 functions simulates the input 3-phases voltages          #
#                                                                            #
#  Input: time in seconds                                                    #
#  Output: 3-phase voltage signal with harmoncis                             #
#                                                                            #
##############################################################################
*/

double In_signal_a(t)
double t;
{
double  first_a,third_a,fifth_a,total_a;

if(t<0.0)
{
first_a=MAG*sin(2.0*PI*60.0*t);
third_a=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60*t+PHASE_SHIFT);
fifth_a=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t);
total_a= first_a+third_a+fifth_a;
}

else if ((t>=0.0)&&(t<0.375))

{
first_a=MAG*sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)-0.4);
third_a=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*0.4
        *cos(4*PI*t)-1.2+PHASE_SHIFT);
fifth_a=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*0.4
        *cos(4*PI*t)-2.0 ); total_a=first_a+third_a+fifth_a;
}

else
{
first_a=MAG*sin(2.0*PI*60.0*t+2.0*PI*0.8*t-2.285);
third_a=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*2.0
        *PI*0.8*t-6.855+PHASE_SHIFT);
fifth_a=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*2.0
*PI*0.8*t-11.425); total_a=first_a+third_a+fifth_a;
}

return(total_a);
}
```

```
double In_signal_b(t)
double t;
{
double  first_b,third_b,fifth_b,total_b;

if(t<0.0)
{

first_b=MAG*sin(2.0*PI*60.0*t+THETA_B);
third_b=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60*t+PHASE_SHIFT);
fifth_b=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t-2.16);
total_b= first_b+third_b+fifth_b;
}


else if ((t>=0.0)&&(t<0.375))
{

first_b=MAG*sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)-0.4+THETA_B);
third_b=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*0.4
        *cos(4*PI*t)-1.2+PHASE_SHIFT);
fifth_b=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*0.4
        *cos(4*PI*t)-2.0-2.16);
total_b=first_b+third_b+fifth_b;
}


else
{
first_b=MAG*sin(2.0*PI*60.0*t+2.0*PI*0.8*t-2.285+THETA_B);
third_b=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*2.0
        *PI*0.8*t-6.855+PHASE_SHIFT);
fifth_b=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*2.0
*PI*0.8*t-13.585); total_b=first_b+third_b+fifth_b;
}

return(total_b);
}



double In_signal_c(t)
double t;
{
double  first_c,third_c,fifth_c,total_c;

if(t<0.0)
{
first_c=MAG*sin(2.0*PI*60.0*t+THETA_C);
third_c=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60*t+PHASE_SHIFT);
fifth_c=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+2.16);
total_c= first_c+third_c+fifth_c;
}

else if ((t>=0.0)&&(t<0.375))
{
```

```
first_c=MAG*sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)-0.4+THETA_C);
third_c=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*0.4
        *cos(4*PI*t)-1.2+PHASE_SHIFT);
fifth_c=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*0.4
        *cos(4*PI*t)-2.0+2.16); total_c=first_c+third_c+fifth_c;
```



```
}

else
{
first_c=MAG*sin(2.0*PI*60.0*t+2.0*PI*0.8*t-2.285+THETA_C);
third_c=THIRD_H*MAG/100.0*sin(3.0*2.0*PI*60.0*t+3.0*2.0
        *PI*0.8*t-6.855+PHASE_SHIFT);
fifth_c=FIFTH_H*MAG/100.0*sin(5.0*2.0*PI*60.0*t+5.0*2.0
        *PI*0.8*t-9.265); total_c=first_c+third_c+fifth_c;
}

return(total_c);

}




/*
##############################################################################
#                                                                            #
#   Main Program Starts                                                      #
#                                                                            #
##############################################################################
*/

main()
{

int     n,M;
double  Buffer_Va[ND_OVER_2],Buffer_Vb[ND_OVER_2],Buffer_Vc[ND_OVER_2];
double  Buffer_Va_cos[ND_OVER_2],Buffer_Va_sin[ND_OVER_2];
double  Buffer_Vb_cos[ND_OVER_2],Buffer_Vb_sin[ND_OVER_2];
double  Buffer_Vc_cos[ND_OVER_2],Buffer_Vc_sin[ND_OVER_2];
double  Func_true_freq(),In_signal_a(),In_signal_b(),In_signal_c();
double  t,freq_est_last,freq_est_current,freq_true,freq_ave,max_df;
double  Va,Vb,Vc;
double  sum_Va_cos,sum_Va_sin,sum_Vb_cos,sum_Vb_sin,sum_Vc_cos,sum_Vc_sin;
double  angle_a, angle_b,angle_c,angle_ave_last,angle_ave_current;



/*
##############################################################################
#                                                                            #
#   Initilization of variables                                               #
#                                                                            #
```

```
#
#################################################################################
*/

M=0;
t=-0.05;
freq_est_last=60.0;
```



```
angle_ave_last=0.0;

printf("hello\n");

/*
#################################################################################
#
#   Initilization of Buffers                                                    #
#                                                                               #
#################################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
Buffer_Va_cos[n]=0.0;
Buffer_Va_sin[n]=0.0;
Buffer_Vb_cos[n]=0.0;
Buffer_Vb_sin[n]=0.0;
Buffer_Vc_cos[n]=0.0;
Buffer_Vc_sin[n]=0.0;
}

/*
#################################################################################
#
#  While Loop                                                                   #
#                                                                               #
#################################################################################
*/


while (t<TIMELIMIT)
{

freq_true=Func_true_freq(t);


/*
#################################################################################
#
#  Get 3-phase voltages (with harmoncis) by calling functions                   #
#                                                                               #
#################################################################################
*/

Va=In_signal_a(t);
Vb=In_signal_b(t);
Vc=In_signal_c(t);
```

```
/*
###############################################################################
#                                                                             #
```



```
#  Shift the correlation product buffers (or window) once                     #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2-1;n++)
{

Buffer_Va_cos[n]=Buffer_Va_cos[n+1];
Buffer_Va_sin[n]=Buffer_Va_sin[n+1];
Buffer_Vb_cos[n]=Buffer_Vb_cos[n+1];
Buffer_Vb_sin[n]=Buffer_Vb_sin[n+1];
Buffer_Vc_cos[n]=Buffer_Vc_cos[n+1];
Buffer_Vc_sin[n]=Buffer_Vc_sin[n+1];
}

if (M==ND) M=0;

/*
###############################################################################
#                                                                             #
#  Calculate the correlation product for the most recent input signals        #
#  by multiplying voltages and the reference cosine and sine curves           #
#                                                                             #
###############################################################################
*/

Buffer_Va_cos[ND_OVER_2-1]=Va*cos(2*PI*(M%ND)/ND);
Buffer_Va_sin[ND_OVER_2-1]=Va*sin(2*PI*(M%ND)/ND);
Buffer_Vb_cos[ND_OVER_2-1]=Vb*cos(2*PI*(M%ND)/ND);
Buffer_Vb_sin[ND_OVER_2-1]=Vb*sin(2*PI*(M%ND)/ND);
Buffer_Vc_cos[ND_OVER_2-1]=Vc*cos(2*PI*(M%ND)/ND);
Buffer_Vc_sin[ND_OVER_2-1]=Vc*sin(2*PI*(M%ND)/ND);

M+=1;


sum_Va_cos=0.0;
sum_Va_sin=0.0;
sum_Vb_cos=0.0;
sum_Vb_sin=0.0;
sum_Vc_cos=0.0;
sum_Vc_sin=0.0;

/*
###############################################################################
#                                                                             #
#  Summation of all the correlation products                                  #
```

```
#                                                                              #
################################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
sum_Va_cos+=Buffer_Va_cos[n];
sum_Va_sin+=Buffer_Va_sin[n];
sum_Vb_cos+=Buffer_Vb_cos[n];
```



```
sum_Vb_sin+=Buffer_Vb_sin[n];
sum_Vc_cos+=Buffer_Vc_cos[n];
sum_Vc_sin+=Buffer_Vc_sin[n];
}

/*
################################################################################
#                                                                              #
# Calculate the phase angles of the 3 phasors                                  #
#                                                                              #
################################################################################
*/

angle_a=atan(sum_Va_cos/sum_Va_sin);
angle_b=atan(sum_Vb_cos/sum_Vb_sin);
angle_c=atan(sum_Vc_cos/sum_Vc_sin);

/*
################################################################################
#                                                                              #
# Find the average of 3 phasores' angles                                       #
#                                                                              #
################################################################################
*/

angle_ave_current=(angle_a+angle_b+angle_c)/3;

freq_est_current=60.0+(angle_ave_current-angle_ave_last)*60.0*ND/2.0/PI;

/*
################################################################################
#                                                                              #
# df/dt limitor                                                                #
#                                                                              #
################################################################################
*/

if(fabs((freq_est_current-freq_est_last))>MAXDF)
freq_est_current=freq_est_last;



printf(" %f  %f %f %f %f %f \n",t,freq_true,freq_est_current,Va,Vb,Vc);


freq_est_last=freq_est_current;
```

```
angle_ave_last=angle_ave_current;
t+=1.0/ND/60.0;
}
}
```

Appendix 3

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program generates random number                             #
#                                                                            #
#  Reference: Numerical Recipes in C                                         #
#                                                                            #
#                                                                            #
##############################################################################
*/


#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

float ran1(idum)
int *idum;
{
 static long ix1,ix2,ix3;
 static float r[98];
 float temp;
 static int iff=0;
 int j;
 void printf();

 if (*idum < 0 || iff == 0) {
         iff=1;
         ix1=(IC1-(*idum)) % M1;
         ix1=(IA1*ix1+IC1) % M1;
         ix2=ix1 % M2;
         ix1=(IA1*ix1+IC1) % M1;
         ix3=ix1 % M3;
         for (j=1;j<=97;j++) {
                 ix1=(IA1*ix1+IC1) % M1;
                 ix2=(IA2*ix2+IC2) % M2;
                 r[j]=(ix1+ix2*RM2)*RM1;
         }
         *idum=1;
 }
 ix1=(IA1*ix1+IC1) % M1;
 ix2=(IA2*ix2+IC2) % M2;
 ix3=(IA3*ix3+IC3) % M3;
 j=1 + ((97*ix3)/M3);
 if (j > 97 || j < 1) printf("RAN1: This cannot happen.");
 temp=r[j];
 r[j]=(ix1+ix2*RM2)*RM1;
 return temp;
```

```
}


#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
#undef M3
#undef IA3
#undef IC3
```

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program generates the Gaussian Distributed Random No.       #
#                                                                            #
#  Reference: Numerical Recipes in C                                         #
#                                                                            #
#                                                                            #
##############################################################################
*/

#include <math.h>

float gasdev(idum)
int *idum;
{
 static int iset=0;
 static float gset;
 float fac,r,v1,v2;
 float ran1();

 if  (iset == 0) {
        do {
                v1=2.0*ran1(idum)-1.0;
                v2=2.0*ran1(idum)-1.0;
                r=v1*v1+v2*v2;
        } while (r >= 1.0);
        fac=sqrt(-2.0*log(r)/r);
        gset=v1*fac;
        iset=1;
        return v2*fac;
 } else {
        iset=0;
        return gset;
 }
}
```

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program estimates the signal frequency using modified       #
#  Discrete Fourier Transform technique.                                     #
#                                                                            #
#  Input:  simulated 3-phases voltages                                       #
#  Output: estimated frequency                                               #
#                                                                            #
#  Signal component: Pure 60 Hz                                              #
#        + Gausiaan Distributed random noise                                 #
#                                                                            #
##############################################################################
*/


#include <stdio.h>              /* header */
#include <math.h>
#include "nr.h"

#define ND              32              /* no. of samples taken per cycle */
#define ND_OVER_2       ND/2            /* half window calculation        */
#define PI              3.141592
#define  MAG            1.0             /* magnitude of the voltage */

#define PHI_A           0.5             /* arbitary angles chosen so that */
#define PHI_B           0.1             /* curves are continuous at       */
#define PHI_C           -1.785          /* transition */


#define THETA_A         0.0                     /* phase angle diff. */
#define THETA_B         THETA_A+2.0*PI/3        /* for 3 phases       */
#define THETA_C         THETA_A+4.0*PI/3

#define TIMELIMIT       0.5                     /*  simulation time   */
#define  MAXDF          10.0/ND/60.0+0.01      /* 10 Hz/sec max.change */


/*
##############################################################################
#                                                                            #
#  Purpose: This function calculate  the true frequency of the input         #
#           signal                                                           #
#                                                                            #
#  Input:  time in seconds                                                   #
#  Output: true frequency                                                    #
#                                                                            #
#  true frequenc = 60.0 Hz              for t < 0                            #
#                = time varying             0<= t  <=0.375 sec.             #
#                =60.8 Hz                   0.375 < t                       #
#                                                                            #
##############################################################################
*/

double Func_true_freq(t)
double t;
{
```

```
double true_freq;

 if(t<0.0)
 true_freq=60.0;
 else if ((t>=0.0)&&(t<0.375))
 true_freq=60.0-0.8*sin(4*PI*t);
 else
 true_freq=60.8;

return(true_freq);
}


/*
##############################################################################
#                                                                            #
#  Purpose: These 3 functions simulates the input 3-phases voltages          #
#                                                                            #
#  Input: time in seconds, exteral random number generating  function        #
#  Output: 3-phase voltage signal + Gaussian random noise                    #
#                                                                            #
##############################################################################
*/

double In_signal_a(t)
double t;
{
double signal;

 if(t<0.0)
 signal=sin(2.0*PI*60.0*t+PHI_A);
 else if ((t>=0.0)&&(t<0.375))
 signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B);
 else
 signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C);

return(signal);
}

double In_signal_b(t)
double t;
{
double signal;

 if(t<0.0)
 signal=sin(2.0*PI*60.0*t+PHI_A+THETA_B);
 else if ((t>=0.0)&&(t<0.375))
 signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B+THETA_B);
 else
 signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C+THETA_B);

return(signal);
}


double In_signal_c(t)
```

```
double t;
{
double signal;

 if(t<0.0)
 signal=sin(2.0*PI*60.0*t+PHI_A+THETA_C);
 else if ((t>=0.0)&&(t<0.375))
 signal=sin(2.0*PI*60.0*t+0.4*cos(4*PI*t)+PHI_B+THETA_C);
 else
 signal=sin(2.0*PI*60.0*t+2*PI*0.8*t+PHI_C+THETA_C);

return(signal);
}


/*
##############################################################################
#                                                                            #
#   Main Program Starts                                                      #
#                                                                            #
##############################################################################
*/



main()
{

int      n,M;
double   Buffer_Va[ND_OVER_2],Buffer_Vb[ND_OVER_2],Buffer_Vc[ND_OVER_2];
double   Buffer_Va_cos[ND_OVER_2],Buffer_Va_sin[ND_OVER_2];
double   Buffer_Vb_cos[ND_OVER_2],Buffer_Vb_sin[ND_OVER_2];
double   Buffer_Vc_cos[ND_OVER_2],Buffer_Vc_sin[ND_OVER_2];
double   Func_true_freq(),In_signal_a(),In_signal_b(),In_signal_c();
double   t,freq_est_last,freq_est_current,freq_true;
double   Va,Vb,Vc;
double   sum_Va_cos,sum_Va_sin,sum_Vb_cos,sum_Vb_sin,sum_Vc_cos,sum_Vc_sin;
double   angle_a, angle_b,angle_c,angle_ave_last,angle_ave_current;
int      idum=(-16);
double   noise_a,noise_b,noise_c,noise;

/*
##############################################################################
#                                                                            #
#   Initilization of variables                                               #
#                                                                            #
##############################################################################
*/

M=0;
t=-0.05;
freq_est_last=60.0;
angle_ave_last=0.0;

printf("hello\n");
```

```
/*
##############################################################################
#                                                                            #
#   Initilization of Buffers                                                 #
#                                                                            #
##############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
Buffer_Va_cos[n]=0.0;
Buffer_Va_sin[n]=0.0;
Buffer_Vb_cos[n]=0.0;
Buffer_Vb_sin[n]=0.0;
Buffer_Vc_cos[n]=0.0;
Buffer_Vc_sin[n]=0.0;
}

/*
##############################################################################
#                                                                            #
#  While Loop                                                                #
#                                                                            #
##############################################################################
*/


while (t<TIMELIMIT)
{

freq_true=Func_true_freq(t);


/*
##############################################################################
#                                                                            #
#   Generate Gaussian random noise by calling external functions,            #
#   gasdev.c                                                                 #
#                                                                            #
##############################################################################
*/

noise_a=(gasdev(&idum))/200.0;
noise_b=(gasdev(&idum))/200.0;
noise_c=(gasdev(&idum))/200.0;



/*same noise for each phase*/

/*
noise=(gasdev(&idum))/20.0;
noise_a=noise;
noise_b=noise;
noise_c=noise;
```

```
*/


/*
##############################################################################
#                                                                            #
#  Get 3-phase voltages by calling functions                                 #
#                                                                            #
##############################################################################
*/

Va=In_signal_a(t)+noise_a;
Vb=In_signal_b(t)+noise_b;
Vc=In_signal_c(t)+noise_c;


/*
##############################################################################
#                                                                            #
#  Shift the correlation product buffers (or window) once                    #
#                                                                            #
##############################################################################
*/

for (n=0;n<ND_OVER_2-1;n++)
{

Buffer_Va_cos[n]=Buffer_Va_cos[n+1];
Buffer_Va_sin[n]=Buffer_Va_sin[n+1];
Buffer_Vb_cos[n]=Buffer_Vb_cos[n+1];
Buffer_Vb_sin[n]=Buffer_Vb_sin[n+1];
Buffer_Vc_cos[n]=Buffer_Vc_cos[n+1];
Buffer_Vc_sin[n]=Buffer_Vc_sin[n+1];
}

if (M==ND) M=0;


/*
##############################################################################
#                                                                            #
#  Calculate the correlation product for the most recent input signals       #
#  by multiplying voltages and the reference cosine and sine curves          #
#                                                                            #
##############################################################################
*/
Buffer_Va_cos[ND_OVER_2-1]=Va*cos(2*PI*(M%ND)/ND);
Buffer_Va_sin[ND_OVER_2-1]=Va*sin(2*PI*(M%ND)/ND);
Buffer_Vb_cos[ND_OVER_2-1]=Vb*cos(2*PI*(M%ND)/ND);
Buffer_Vb_sin[ND_OVER_2-1]=Vb*sin(2*PI*(M%ND)/ND);
Buffer_Vc_cos[ND_OVER_2-1]=Vc*cos(2*PI*(M%ND)/ND);
Buffer_Vc_sin[ND_OVER_2-1]=Vc*sin(2*PI*(M%ND)/ND);

M+=1;


sum_Va_cos=0.0;
```

```
sum_Va_sin=0.0;
sum_Vb_cos=0.0;
sum_Vb_sin=0.0;
sum_Vc_cos=0.0;
sum_Vc_sin=0.0;

/*
###############################################################################
#                                                                             #
#  Summation of all the correlation products                                  #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
sum_Va_cos+=Buffer_Va_cos[n];
sum_Va_sin+=Buffer_Va_sin[n];
sum_Vb_cos+=Buffer_Vb_cos[n];
sum_Vb_sin+=Buffer_Vb_sin[n];
sum_Vc_cos+=Buffer_Vc_cos[n];
sum_Vc_sin+=Buffer_Vc_sin[n];
}

/*
###############################################################################
#                                                                             #
# Calculate the phase angles of the 3 phasors                                 #
#                                                                             #
###############################################################################
*/

angle_a=atan(sum_Va_cos/sum_Va_sin);
angle_b=atan(sum_Vb_cos/sum_Vb_sin);
angle_c=atan(sum_Vc_cos/sum_Vc_sin);

/*
###############################################################################
#                                                                             #
# Find the average of 3 phasores' angles                                      #
#                                                                             #
###############################################################################
*/

angle_ave_current=(angle_a+angle_b+angle_c)/3;

freq_est_current=60.0+(angle_ave_current-angle_ave_last)*60.0*ND/2.0/PI;


/*
###############################################################################
#                                                                             #
# df/dt limitor                                                               #
#                                                                             #
###############################################################################
*/
```

```
if(fabs((freq_est_current-freq_est_last))>MAXDF)
freq_est_current=freq_est_last;


printf(" %f %f %f %f %f %f %f %f %f\n",t,freq_true,freq_est_current,Va,Vb,Vc
       ,noise_a,noise_b,noise_c);


freq_est_last=freq_est_current;
angle_ave_last=angle_ave_current;
t+=1.0/ND/60.0;
}
}
```

Appendix 4

```
/*
##############################################################################
#                                                                            #
#  Purpose: This program estimates the signal frequency using modified       #
#  Discrete Fourier Transform technique.                                     #
#                                                                            #
#  Input:  simulated 3-phases voltages                                       #
#  Output: estimated frequency                                               #
#                                                                            #
#  Signal component: Pure 60 Hz signal with -90 degree                       #
#                    phase shift over one cycle                              #
#                                                                            #
##############################################################################
*/

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


#define ND              32              /* no. of samples taken per cycle */
#define ND_OVER_2       ND/2            /* half window calculation        */
#define PI              3.141592
#define  MAG            1.0             /* magnitude of the voltage */


#define PHI_A           0.0             /* arbitary angles chosen so that */
#define PHI_B           0.0             /* curves are continuous at       */
#define PHI_C           0.0             /* transition */


#define THETA_A         0.0                     /* phase angle diff. */
#define THETA_B         THETA_A+2.0*PI/3        /* for 3 phases      */
#define THETA_C         THETA_A+4.0*PI/3

#define TIMELIMIT       0.5                     /*  simulation time   */
#define  MAXDF          10.0/ND/60.0+0.01        /* 10 Hz/sec max.change */


/*
##############################################################################
#                                                                            #
#  Purpose: This function calculate  the true frequency of the input         #
#           signal                                                           #
#                                                                            #
#  Input:  time in seconds                                                   #
#  Output: true frequency                                                    #
#                                                                            #
#  true frequenc = 60.0 Hz                                                   #
#                                                                            #
##############################################################################
*/
double Func_true_freq(t)
double t;
{
double true_freq;
```

```
 if(t<0.0)
 true_freq=60.0;
 else if ((t>=0.0)&&(t<1.0/60.0))
 true_freq=45.0;
 else
 true_freq=60.0;

return(true_freq);
}

/*
##############################################################################
#                                                                            #
#  Purpose: These 3 functions simulates the input 3-phases voltages          #
#                                                                            #
#  Input: time in seconds                                                    #
#  Output: 3-phase voltage signal (90deg. phase shift over 1 cycle)          #
#                                                                            #
##############################################################################
*/
double In_signal_a(t)
double t;
{
double signal;

 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A);
 else if ((t>=0.0)&&(t<1.0/60.0))
 signal=MAG*sin(2.0*PI*60.0*t-30.0*PI*t+PHI_B);
 else
 signal=MAG*sin(2.0*PI*60.0*t-PI/2.0+PHI_C);

return(signal);
}

double In_signal_b(t)
double t;
{
double signal;

 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A+THETA_B);
 else if ((t>=0.0)&&(t<1.0/60.0))
 signal=MAG*sin(2.0*PI*60.0*t-30.0*PI*t+PHI_B+THETA_B);
 else
 signal=MAG*sin(2.0*PI*60.0*t-PI/2.0+PHI_C+THETA_B);

return(signal);
}


double In_signal_c(t)
double t;
{
double signal;
```

```
 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A+THETA_C);
 else if ((t>=0.0)&&(t<1.0/60.0))
 signal=MAG*sin(2.0*PI*60.0*t-30.0*PI*t+PHI_B+THETA_C);
 else
 signal=MAG*sin(2.0*PI*60.0*t-PI/2.0+PHI_C+THETA_C);

return(signal);
}



/*
##############################################################################
#                                                                            #
#   Main Program Starts                                                      #
#                                                                            #
##############################################################################
*/
main()
{

int      n,M;
double   Buffer_Va[ND_OVER_2],Buffer_Vb[ND_OVER_2],Buffer_Vc[ND_OVER_2];
double   Buffer_Va_cos[ND_OVER_2],Buffer_Va_sin[ND_OVER_2];
double   Buffer_Vb_cos[ND_OVER_2],Buffer_Vb_sin[ND_OVER_2];
double   Buffer_Vc_cos[ND_OVER_2],Buffer_Vc_sin[ND_OVER_2];
double   Func_true_freq(),In_signal_a(),In_signal_b(),In_signal_c();
double   t,freq_est_last,freq_est_current,freq_true;
double   Va,Vb,Vc;
double   sum_Va_cos,sum_Va_sin,sum_Vb_cos,sum_Vb_sin,sum_Vc_cos,sum_Vc_sin;
double   angle_a, angle_b,angle_c,angle_ave_last,angle_ave_current;


/*
##############################################################################
#                                                                            #
#   Initilization of variables                                               #
#                                                                            #
##############################################################################
*/

M=0;
t=-0.05;
freq_est_last=60.0;
angle_ave_last=0.0;

printf("hello\n");

/*
##############################################################################
#                                                                            #
#   Initilization of Buffers                                                 #
#                                                                            #
```

```
################################################################################
*/
for (n=0;n<ND_OVER_2;n++)
{
Buffer_Va_cos[n]=0.0;
Buffer_Va_sin[n]=0.0;
Buffer_Vb_cos[n]=0.0;
Buffer_Vb_sin[n]=0.0;
Buffer_Vc_cos[n]=0.0;
Buffer_Vc_sin[n]=0.0;
}

/*
################################################################################
#                                                                              #
#  While Loop                                                                  #
#                                                                              #
################################################################################
*/

while (t<TIMELIMIT)
{

freq_true=Func_true_freq(t);


/*
################################################################################
#                                                                              #
#  Get 3-phase voltages by calling functions                                   #
#                                                                              #
################################################################################
*/

Va=In_signal_a(t);
Vb=In_signal_b(t);
Vc=In_signal_c(t);


/*
################################################################################
#                                                                              #
#  Shift the correlation product buffers (or window) once                      #
#                                                                              #
################################################################################
*/

for (n=0;n<ND_OVER_2-1;n++)
{

Buffer_Va_cos[n]=Buffer_Va_cos[n+1];
Buffer_Va_sin[n]=Buffer_Va_sin[n+1];
Buffer_Vb_cos[n]=Buffer_Vb_cos[n+1];
Buffer_Vb_sin[n]=Buffer_Vb_sin[n+1];
Buffer_Vc_cos[n]=Buffer_Vc_cos[n+1];
Buffer_Vc_sin[n]=Buffer_Vc_sin[n+1];
```

```
}

if (M==ND) M=0;

/*
###############################################################################
#                                                                             #
#  Calculate the correlation product for the most recent input signals        #
#  by multiplying voltages and the reference cosine and sine curves           #
#                                                                             #
###############################################################################
*/

Buffer_Va_cos[ND_OVER_2-1]=Va*cos(2*PI*(M%ND)/ND);
Buffer_Va_sin[ND_OVER_2-1]=Va*sin(2*PI*(M%ND)/ND);
Buffer_Vb_cos[ND_OVER_2-1]=Vb*cos(2*PI*(M%ND)/ND);
Buffer_Vb_sin[ND_OVER_2-1]=Vb*sin(2*PI*(M%ND)/ND);
Buffer_Vc_cos[ND_OVER_2-1]=Vc*cos(2*PI*(M%ND)/ND);
Buffer_Vc_sin[ND_OVER_2-1]=Vc*sin(2*PI*(M%ND)/ND);

M+=1;


sum_Va_cos=0.0;
sum_Va_sin=0.0;
sum_Vb_cos=0.0;
sum_Vb_sin=0.0;
sum_Vc_cos=0.0;
sum_Vc_sin=0.0;

/*
###############################################################################
#                                                                             #
#  Summation of all the correlation products                                  #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
sum_Va_cos+=Buffer_Va_cos[n];
sum_Va_sin+=Buffer_Va_sin[n];
sum_Vb_cos+=Buffer_Vb_cos[n];
sum_Vb_sin+=Buffer_Vb_sin[n];
sum_Vc_cos+=Buffer_Vc_cos[n];
sum_Vc_sin+=Buffer_Vc_sin[n];
}


/*
###############################################################################
#                                                                             #
# Calculate the phase angles of the 3 phasors                                 #
#                                                                             #
###############################################################################
*/
angle_a=atan(sum_Va_cos/sum_Va_sin);
```

```
angle_b=atan(sum_Vb_cos/sum_Vb_sin);
angle_c=atan(sum_Vc_cos/sum_Vc_sin);

/*
###############################################################################
#                                                                             #
# Find the average of 3 phasores' angles                                      #
#                                                                             #
###############################################################################
*/

angle_ave_current=(angle_a+angle_b+angle_c)/3;

/*
###############################################################################
#                                                                             #
# Estimate the frequency of input signal                                      #
#                                                                             #
###############################################################################
*/
freq_est_current=60.0+(angle_ave_current-angle_ave_last)*60.0*ND/2.0/PI;

if (fabs((freq_est_current-freq_est_last))>MAXDF)
freq_est_current=freq_est_last;



printf(" %f  %f %f %f %f %f \n",t,freq_true,freq_est_current,Va,Vb,Vc);


freq_est_last=freq_est_current;
angle_ave_last=angle_ave_current;
t+=1.0/ND/60.0;
}
}
```

Appendix 5

```
/*
###############################################################################
#                                                                             #
#  Purpose: This program estimates the signal frequency using modified        #
#  Discrete Fourier Transform technique.                                      #
#           .                                                                 #
#  Input:  simulated 3-phases voltages                                        #
#  Output: estimated frequency                                                #
#                                                                             #
#  Signal component: Pure 60 Hz signal with 50% magnitude drop                #
#                    at time =0                                               #
#                                                                             #
###############################################################################
*/


#include <stdio.h>              /* header */
#include <math.h>


#define ND              32              /* no. of samples taken per cycle */
#define ND_OVER_2       ND/2            /* half window calculation        */
#define PI              3.141592
#define  MAG            1.0             /* magnitude of the voltage */


#define PHI_A           0.5             /* arbitary angles chosen so that */
#define PHI_B           1.28            /* curves are continuous at       */
#define PHI_C           1.28            /* transition */


#define THETA_A         0.0                     /* phase angle diff. */
#define THETA_B         THETA_A+2.0*PI/3        /* for 3 phases      */
#define THETA_C         THETA_A+4.0*PI/3

#define TIMELIMIT       0.5                     /*  simulation time   */
#define  MAXDF          10.0/ND/60.0+0.01        /* 10 Hz/sec max.change */


/*
###############################################################################
#                                                                             #
#  Purpose: This function calculate  the true frequency of the input          #
#           signal                                                            #
#                                                                             #
#  Input:  time in seconds                                                    #
#  Output: true frequency                                                     #
#                                                                             #
#  true frequenc = 60.0 Hz                                                    #
#                                                                             #
###############################################################################
*/
double Func_true_freq(t)
double t;
{
```

```
double true_freq;

 if(t<0.0)
 true_freq=60.0;
 else if ((t>=0.0)&&(t<0.375))
 true_freq=60.0;
 else
 true_freq=60.0;

return(true_freq);
}

/*
##############################################################################
#                                                                            #
#  Purpose: These 3 functions simulates the input 3-phases voltages          #
#                                                                            #
#  Input: time in seconds                                                    #
#  Output: 3-phase voltage signal (50% magnitude drop at t=0)                #
#                                                                            #
##############################################################################
*/

double In_signal_a(t)
double t;
{
double signal;

 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A);
 else if ((t>=0.0)&&(t<0.375))
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_B);
 else
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_C);

return(signal);
}

double In_signal_b(t)
double t;
{
double signal;

 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A+THETA_B);
 else if ((t>=0.0)&&(t<0.375))
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_B+THETA_B);
 else
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_C+THETA_B);

return(signal);
}


double In_signal_c(t)
double t;
```

```
{
double signal;

 if(t<0.0)
 signal=MAG*sin(2.0*PI*60.0*t+PHI_A+THETA_C);
 else if ((t>=0.0)&&(t<0.375))
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_B+THETA_C);
 else
 signal=MAG/2.0*sin(2.0*PI*60.0*t+PHI_C+THETA_C);

return(signal);
}



/*
##############################################################################
#                                                                            #
#   Main Program Starts                                                      #
#                                                                            #
##############################################################################
*/

main()
{

int       n,M;
double    Buffer_Va[ND_OVER_2],Buffer_Vb[ND_OVER_2],Buffer_Vc[ND_OVER_2];
double    Buffer_Va_cos[ND_OVER_2],Buffer_Va_sin[ND_OVER_2];
double    Buffer_Vb_cos[ND_OVER_2],Buffer_Vb_sin[ND_OVER_2];
double    Buffer_Vc_cos[ND_OVER_2],Buffer_Vc_sin[ND_OVER_2];
double    Func_true_freq(),In_signal_a(),In_signal_b(),In_signal_c();
double    t,freq_est_last,freq_est_current,freq_true,max_df;
double    Va,Vb,Vc;
double    sum_Va_cos,sum_Va_sin,sum_Vb_cos,sum_Vb_sin,sum_Vc_cos,sum_Vc_sin;
double    angle_a, angle_b,angle_c,angle_ave_last,angle_ave_current;



/*
##############################################################################
#                                                                            #
#   Initilization of variables                                               #
#                                                                            #
##############################################################################
*/

M=0;
t=-0.05;
freq_est_last=60.0;
angle_ave_last=0.0;

printf("hello\n");

/*
```

```
/*
###############################################################################
#                                                                             #
#   Initilization of Buffers                                                  #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
Buffer_Va_cos[n]=0.0;
Buffer_Va_sin[n]=0.0;
Buffer_Vb_cos[n]=0.0;
Buffer_Vb_sin[n]=0.0;
Buffer_Vc_cos[n]=0.0;
Buffer_Vc_sin[n]=0.0;
}


/*
###############################################################################
#                                                                             #
#  While Loop                                                                 #
#                                                                             #
###############################################################################
*/

while (t<TIMELIMIT)
{

freq_true=Func_true_freq(t);


/*
###############################################################################
#                                                                             #
#  Get 3-phase voltages by calling functions                                  #
#                                                                             #
###############################################################################
*/

Va=In_signal_a(t);
Vb=In_signal_b(t);
Vc=In_signal_c(t);


/*
###############################################################################
#                                                                             #
#  Shift the correlation product buffers (or window) once                     #
#                                                                             #
###############################################################################
*/

for (n=0;n<ND_OVER_2-1;n++)
{
```

```
Buffer_Va_cos[n]=Buffer_Va_cos[n+1];
Buffer_Va_sin[n]=Buffer_Va_sin[n+1];
Buffer_Vb_cos[n]=Buffer_Vb_cos[n+1];
Buffer_Vb_sin[n]=Buffer_Vb_sin[n+1];
Buffer_Vc_cos[n]=Buffer_Vc_cos[n+1];
Buffer_Vc_sin[n]=Buffer_Vc_sin[n+1];
}

if (M==ND) M=0;

/*
##############################################################################
#                                                                            #
#  Calculate the correlation product for the most recent input signals       #
#  by multiplying voltages and the reference cosine and sine curves          #
#                                                                            #
##############################################################################
*/

Buffer_Va_cos[ND_OVER_2-1]=Va*cos(2*PI*(M%ND)/ND);
Buffer_Va_sin[ND_OVER_2-1]=Va*sin(2*PI*(M%ND)/ND);
Buffer_Vb_cos[ND_OVER_2-1]=Vb*cos(2*PI*(M%ND)/ND);
Buffer_Vb_sin[ND_OVER_2-1]=Vb*sin(2*PI*(M%ND)/ND);
Buffer_Vc_cos[ND_OVER_2-1]=Vc*cos(2*PI*(M%ND)/ND);
Buffer_Vc_sin[ND_OVER_2-1]=Vc*sin(2*PI*(M%ND)/ND);

M+=1;


sum_Va_cos=0.0;
sum_Va_sin=0.0;
sum_Vb_cos=0.0;
sum_Vb_sin=0.0;
sum_Vc_cos=0.0;
sum_Vc_sin=0.0;

/*
##############################################################################
#                                                                            #
#  Summation of all the correlation products                                 #
#                                                                            #
##############################################################################
*/

for (n=0;n<ND_OVER_2;n++)
{
sum_Va_cos+=Buffer_Va_cos[n];
sum_Va_sin+=Buffer_Va_sin[n];
sum_Vb_cos+=Buffer_Vb_cos[n];
sum_Vb_sin+=Buffer_Vb_sin[n];
sum_Vc_cos+=Buffer_Vc_cos[n];
sum_Vc_sin+=Buffer_Vc_sin[n];
}

/*
##############################################################################
#                                                                            #
```

```
# Calculate the phase angles of the 3 phasors                                    #
#                                                                                #
##################################################################################
*/

angle_a=atan(sum_Va_cos/sum_Va_sin);
angle_b=atan(sum_Vb_cos/sum_Vb_sin);
angle_c=atan(sum_Vc_cos/sum_Vc_sin);

/*
##################################################################################
#
# Find the average of 3 phasores' angles                                         #
#                                                                                #
##################################################################################
*/

angle_ave_current=(angle_a+angle_b+angle_c)/3;

/*
##################################################################################
#
# Estimate the frequency of input signal                                         #
#                                                                                #
##################################################################################
*/

freq_est_current=60.0+(angle_ave_current-angle_ave_last)*60.0*ND/2.0/PI;

/*
##################################################################################
#
# df/dt limitor                                                                  #
#                                                                                #
##################################################################################
*/
if(fabs((freq_est_current-freq_est_last))>MAXDF)
freq_est_current=freq_est_last;



printf("   %f %f %f %f %f %f\n",t,freq_true,freq_est_current,Va,Vb,Vc);


freq_est_last=freq_est_current;
angle_ave_last=angle_ave_current;
t+=1.0/ND/60.0;
}
}
```