High Impedance Arcing Faults Detection Using An Artificial Neural Network

A Thesis submitted to

The Faculty of Graduate Studies

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Ahmad Fathi Sultan

B.Sc. (honors), Ain–Shams University, Egypt, 1983

M.Sc. , Ain–Shams University, Egypt, 1987

Department of Electrical and Computer Engineering

The University of Manitoba,

Winnipeg, Manitoba

CANADA

February, 1992

ISBN 0-315-77715-X

Canada

HIGH IMPEDANCE ARCING FAULTS DETECTION USING AN ARTIFICIAL NEURAL NETWORK

BY

AHMAD FATHI SULTAN

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

DOCTOR OF PHILOSOPHY

Abstract

A high impedance fault (HIF) on a power system distribution line could be due to a downed conductor, and is a dangerous situation because the current may be too small to be detected by conventional means. An energized conductor lying on ground threatens human and live–stock life and could result in property damage. Such faults do not cause a major problem for the whole integrity of the electric power system; however, the protection aspect against HIF comes mainly from a moral point of view: safety of utilities' customers and personnel.

The University of Manitoba (UM) and Manitoba Hydro (MH) started problem investigation by conducting staged HIF field tests on a 15 kV distribution line near Sperling, Manitoba. A high voltage laboratory setup was devised, at UM, to collect high impedance fault data under controlled conditions. The laboratory model results were in agreement not only with the field test results but also with data published in the literature by other institutions and research groups. The model was validated. The physics underlying the arcing phenomenon was studied.

The ability of HIF relays to not operate under no–fault conditions (*security*) was examined. Waveforms of high impedance faults as well as loads that behave or appear like high impedance faults were collected and processed.

The detection parameters used in a number of existing high impedance fault detection algorithms were extracted. The algorithms' ability to discriminate between faults and fault–like loads was tested. It was found that there are loads that imitate high impedance faults, and that some of the existing detection algorithms lack the *security* required under these load conditions.

The goal of the study was to develop a *reliable* prototype HIF detector capable of providing the required relay *dependability* (ability to trip when it should), and *security* (ability to **not** trip when it shouldn't).

High impedance fault detection methods were reviewed. No single detection method can detect all electrical conditions resulting from downed conductor faults. Quite a few of the detection methods require extensive computations in the preprocessing stage to extract the features of the input signal(s). A criterion is then applied to obtain the detection parameters.

A feed–forward three–layer artificial neural network (ANN) was trained by high impedance fault, fault–like, and normal load current patterns, using the back–propagation training algorithm. The neural network parameters were embodied in a high impedance arcing fault detection algorithm. Line current zero–crossing, and the width of current conduction period per–cycle are the preprocessing required by the algorithm. The algorithm was tested by traces of normal load current disturbed by currents of faults on dry and wet grassy soil, arc welder, computer, fluorescent light, and sinusoidal loads. The outcomes of this study indicated that the neural network was able to reach a general solution of the problem, for the available training patterns. The detector was able to identify fault events distorted by arcing noise. It was also able to identify the no–fault conditions.

The study suggests that further training with fault and load patterns would increase the robustness of the novel detector; other detection methods operating in parallel with this algorithm would increase the relay reliability; practical implementation would confirm the study results, and could propose some modification to the detection algorithm(s) and/or the pattern recognition neural network to suit the actual power system conditions.

Acknowledgements

I wish to express my deep appreciation to Prof. Glenn W. Swift and Mr. David J. Fedirchuck for suggesting the topic of this project. I am indebted to Prof. Swift for his guidance, council, and encouragement during the course of this study.

The author would like to acknowledge the financial support provided by the Natural Sciences and Engineering Research Council (NSERC) and the research committee of Manitoba Hydro (MH). I am also grateful to NSERC and MH for the assistance with field measurements, and the field support from Mr. David J. Fedirchuck, Mr. Karl Hesse, and Mr. Paul Wilson, all of Manitoba Hydro. I wish to thank the University of Manitoba supporting staff for facilitating the experimental study; particularly Mr. Erwin Dirks for his help in the field and laboratory tests, and Mr. W. Bourbonnais for building and wiring the high voltage laboratory setup. Thanks, also, are to Mr. Adi Indrayanto for his discussions on neural networks.

Finally, I am particularly grateful to my parents, wife, and daughter for their unwavering support and encouragement.

Dedication

To

my wife Amani

&

daughter Aesha

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

Symbol	*Meaning*
A	ampere
A/D	analog–to–digital
A/div	ampere–per–division
AAF	anti–aliasing filter
ANN (or NN)	artificial neural network (or neural network)
BW	band width
ct	current transformer
DSP	digital signal processor
dB	decibel
dc	direct current
EMTP	Electro Magnetic Transient Program
F, μF, pF	farad, micro–farad, pico–farad
FFT	Fast Fourier Transform
HIF	high impedance fault
HIFLL	high impedance fault–like load
Hz, kHz, MHz	hertz, kilo–hertz, mega–hertz
IPS	information processing system
km	kilometers
m, μm	meter, micro–meter

m/s	meter–per–second
pt	potential transformer
pu, PU	per–unit
OCR	oil circuit recloser
Ω, kΩ, MΩ, GΩ	ohm, kilo–ohm, mega–ohm, gega–ohm
RC	resistor–capacitor
RHS	right–hand–side
s, ms	second, milli–second
UVR	ultra–violet recorder
V, mV, kV	volt, milli–volt, kilo–volt
VA, kVA, MVA	volt–ampere, kilo–VA, mega–VA
V/div	volt–per–division
$V_{LL,LN,pp}$	volt line–to–line, line–to–neutral, peak–to–peak
Z_1, Z_2 , Z_o	positive, negative and zero sequence impedances

1 Introduction

1.1. Statement of the Problem [1][2][3][4]

Public safety and service continuity have been the hallmarks of utilities operations since the beginning of widespread electric energy distribution. Risk of public contact with energized conductors will continue as long as power delivery exists. Human contact with an energized conductor can cause injury or death.

1.1.1. Effects of Electricity on Humans

The human body, particularly the heart, is very vulnerable to electrical current. Muscle contraction or paralysis, heart stoppage, and skin burns can result from current flow through the body. These effects depend upon the amount of current, length of time, and current path. Figure 1.1 illustrates how skin resistance and conductor voltage affect the amount of current flow, and the effects of these currents on humans. When a person touches a downed power line, current flows through his body to ground. Depending upon how contact is made, some of the current may flow through his heart. Skin provides a resistance of 1500 to 5000 ohms. Skin puncture, resulting from electric current flow burns, reduces the resistance to as low as 500 ohms. A contact with a 7200 V conductor permits a current flow of

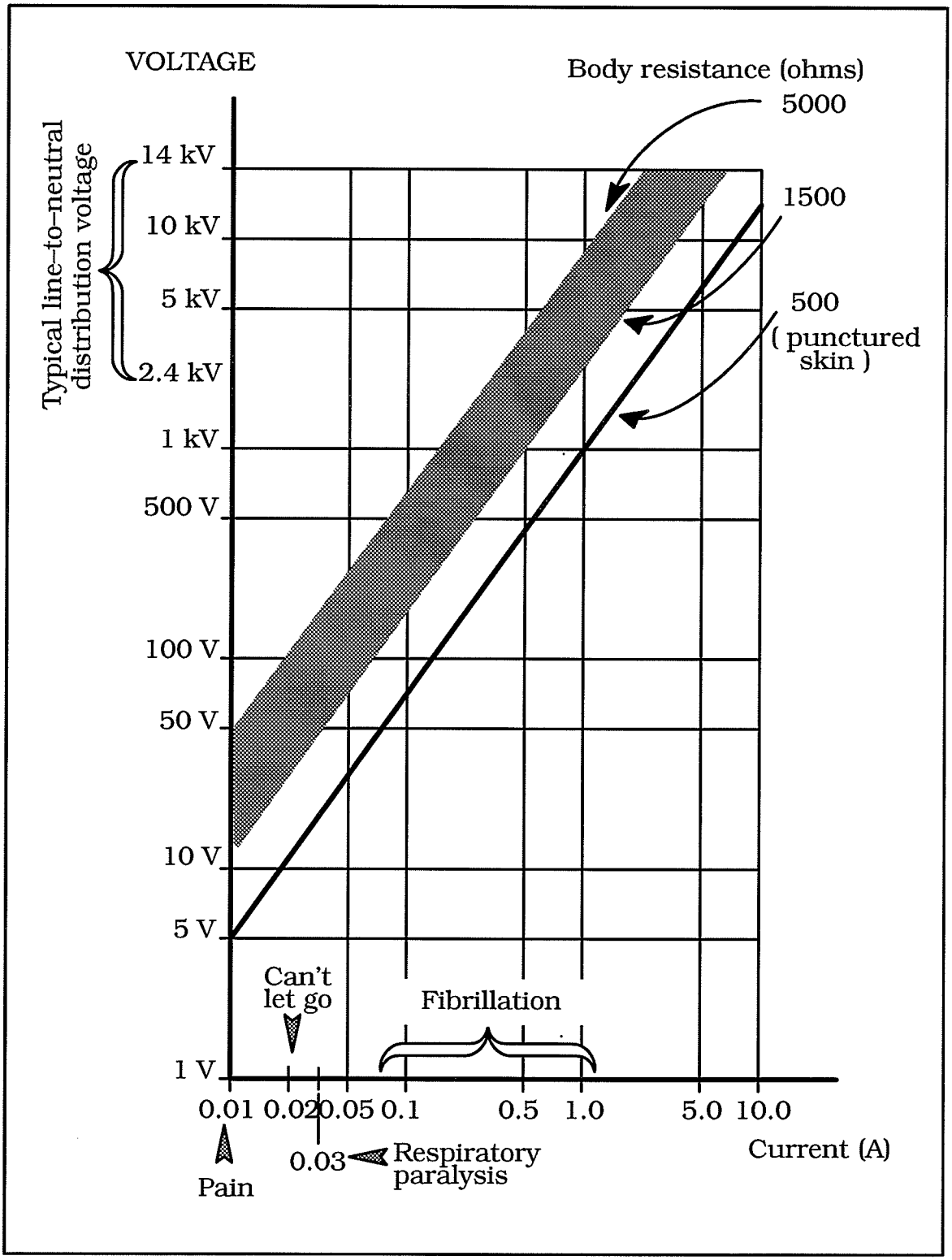

Figure 1.1. Effects of electricity on humans [1].

approximately 1.4 to 4.8 A. Ventricular fibrillation or cardiac standstill occurs in less than one–sixtieth of a second.

1.1.2. Distribution System Faults

High Current Faults

The most common distribution system fault is the short circuit. This type of fault involves contact of a phase conductor(s) and the neutral return, or phase conductors together. The excessive current, resulting from short circuit faults, pushes the transmission lines above their thermal capabilities. Possible destruction to the system could occur if the fault is not cleared. Protection of existing power systems against short circuits is done by relays and fuses.

Low Current Faults

Faults on a distribution system, not involving normal power carrying conductors, could produce an undetectable change in current flow in the circuit. This type of fault is called a high impedance fault (HIF); the impedance at the point of fault is high enough to limit the current flow to a normal load value rather than to a fault level. High impedance faults often exhibit arcing phenomena when no solid return path for the current is available. These faults do not pose a direct threat to the whole integrity of the power system. On the other hand, they present a source of threat to utilities' customers and personnel. Fire hazard, equipment damage, waste of energy, service interruption, and the ensuing utility liability are results of such faults. Figure 1.2 shows the damage resulting from the arcing of a downed conductor on a wooden pallet. Protection against high impedance faults comes mainly from a moral point of view, i.e. improving safety to persons.

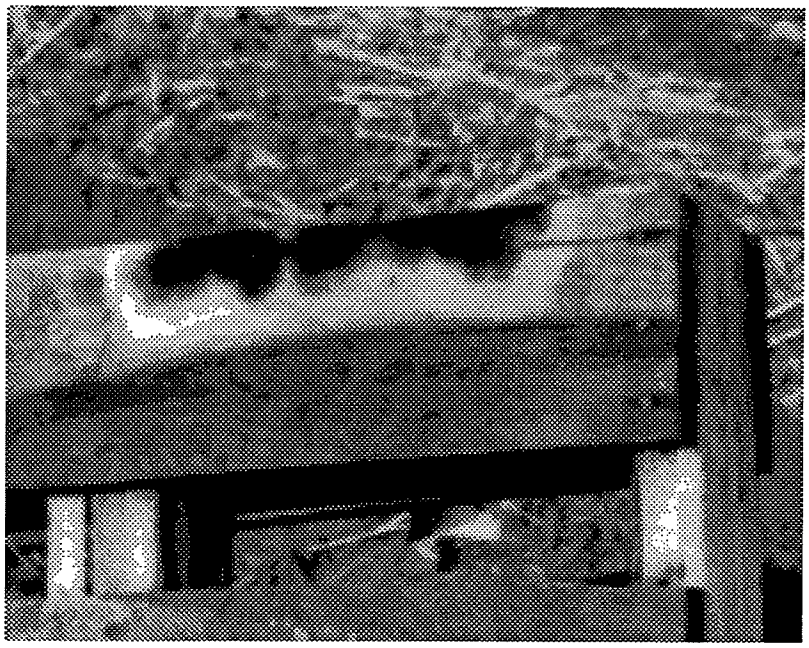

Figure 1.2. Damage resulting from arcing faults to a wooden pallet.

Frequent causes of high impedance, or downed conductor, faults are contact with trees, excessive ice loading, vehicle collisions, people, and contact of covered high voltage conductors with ground (e.g. earth) when a pin-

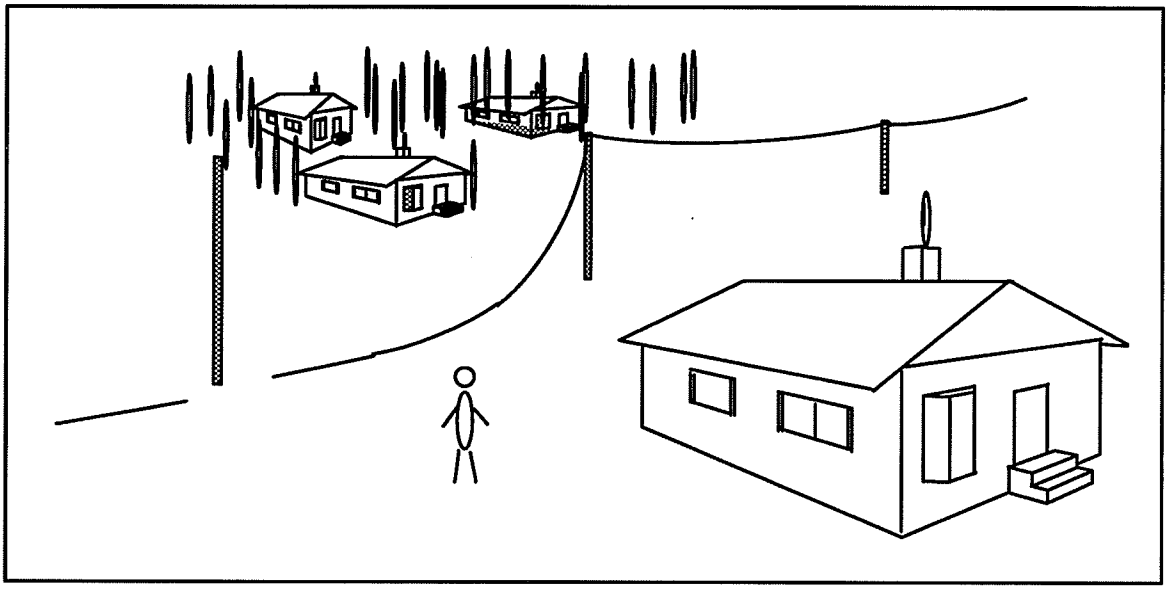

Figure 1.3. Scenario of a high impedance fault.

hole break occurs in the conductor cover. A typical scenario of a HIF is shown in Fig. 1.3 where a conductor of a distribution feeder has separated and fallen on the earth surface below.

1.1.3. Fault Detection

The difficulty in detecting HIF, by conventional short circuit protection devices, could be understood from the relationship of HIF current to overcurrent device settings shown in Fig. 1.4. For a 1 per–unit maximum anticipated full load, the phase–relaying (overcurrent protection) is set at a reasonable increment above the expected full load level: usually 125 to 200 % of rated line current. Undetected HIF currents lie in the region unprotected by overcurrent devices. Detection of HIF by lowering the phase–relaying setting, say to operate at 75 to 125 % of expected load current level, would result in frequent, unnecessary service interruption. Since downed conductor faults result from the contact of a single conductor to a high impedance

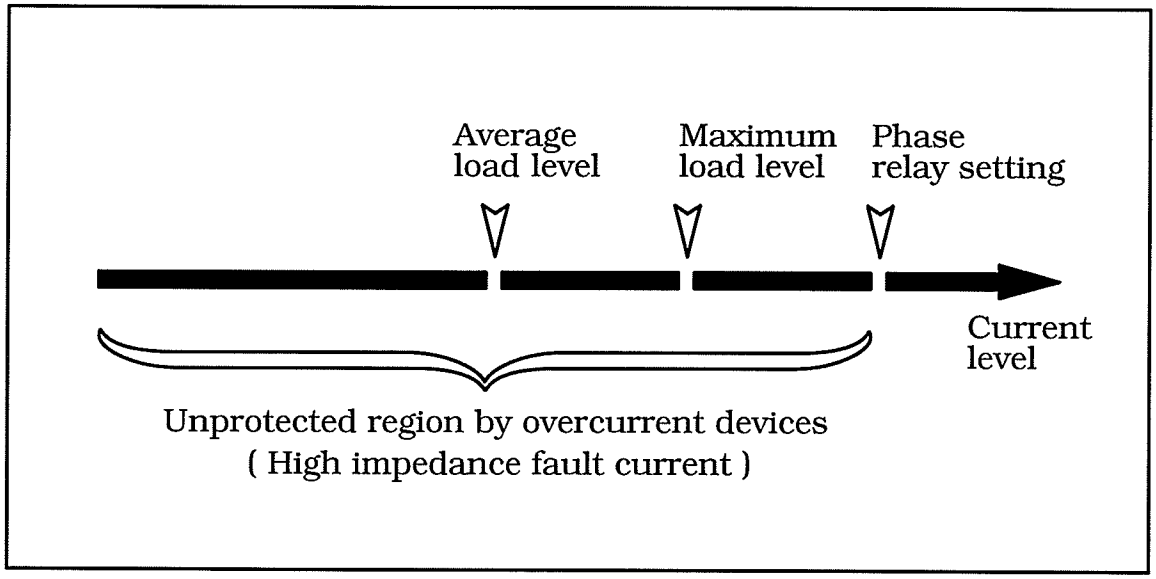

Figure 1.4. Relation of high impedance fault current to overcurrent device settings.

grounded surface, e.g. earth, the use of phase imbalance protection (ground–relaying) would shrink the unprotected region by overcurrent devices down to 25 – 50 % of the phase–relaying setting. Sensitive ground–relaying, however, is an unreliable means of protection against HIF for systems with a high degree of imbalance.

High impedance faults could occur in utility systems as well as in industrial power systems. The occurrence of such faults is common at voltages such as 13.8 kV. The problem is even more acute at 2.4 to 4.8 kV because the fault current magnitude decreases. Its incidence at distribution voltages above 20 kV is low; however, downed conductors have been reported at 69 and 115 kV.

1.2. Previous Research Efforts in Arcing Faults Detection

Arcing high impedance faults are downed conductor faults associated with arcing. An amount of fault current flows into the system.

***Carr** [5]*

A theoretical analysis by Carr showed that, on a grounded–Y–connected system, a signal dependent only on the fault could be calculated by summing fixed proportions of the measured neutral and ground currents. The proportioning constants were ratios of the line and ground impedances. If the individual impedances changed due to ambient conditions, he suggested a feedback path to automatically adjust the proportioning constants.

Broken conductors could be detected by monitoring the sequence voltages at the load side of the line. The negative sequence voltage was prefer-

able to the zero sequence voltage. The former is less dependent on the zero sequence impedance which is dependent on grounding resistance. Communication links would be required, from the detection locations back to the substation, to de–energize the faulted line.

Calhoun et al *[6]*

The ratio–ground–relay is a ground relay whose pickup level varies inversely proportionally to the feeder loading. Since the detection was based on the imbalance of line currents, the detection of fallen and broken conductors was possible.

The relay was commercially introduced in 1981. Later on, utilities which installed the relay had to either completely remove it, or to put it in an alarm mode. The problem of maintaining phase balance caused an unacceptable number of false trips. Some causes of imbalance were single–phase switching, and blowing capacitor fuses. The use of this relay, even as a monitor, required frequent phase current balancing, aggressive tree trimming schedules, and a very structured and effective operator response.

Texas A&M University *[7][8][9][10][11][12]*

The research at Texas A&M University has played a leading role in the detection of arcing high impedance faults using harmonic components of the fault current.

- In a faulted system, a persistent increase in the high frequency components (above 2 kHz) of the current over that of an unfaulted system was observed [7]. This high frequency component is due to several strikes and restrikes in the air gaps, between the conductor and the high impedance surface, prior to a stable arc being established. The Fast Fourier Transform

(FFT) of the high frequency components, spikes, of the current over one 60 Hz cycle period takes a $\frac{\sin x}{x}$ distribution. A spectral estimate, calculated as the average of each spectral line over a 300 power frequency cycle record, was found to be unique for an arcing fault. The frequency domain spectrum was approximately inversely proportional to the frequency. Normal system noisy loads would cause certain frequencies to be displayed in their spectral estimate rather than the smooth frequency distribution of the arcing faults. It was suggested that this method could be used to improve system security against false trips by running it in the background of a fault detector that implements more than a detection algorithm.

- In practical implementations [7][8], the summation of three feeder phase currents was conditioned to sufficiently reject the 60 Hz signal while also amplifying the 2 to 10 kHz components to a measurable level. The filtered signal was sampled at the rate of 64 samples per–cycle. The raw data were summed over an entire 60 Hz cycle. The average of these 64 data points was referred to as the "energy" contained at any time in the high frequency signal relative to the normal system signal. This method was named the "energy algorithm". The term "energy" was **not** used as an absolute measure of energy rather than a figure of merit that indicates the cumulative effect of arc noise. When this average showed a 50 % increase for three out of five consecutive cycles an event was detected. If 48 of the 260 cycles after an event indicated a high frequency above threshold, the event was classified as a fault. The threshold was adapted with the base high frequency signal level on the feeder.

- Field testing of this algorithm [7][9] showed that, for a given relay settings, the relay effectiveness was much reduced if the level of the high frequency component of the current was not enough to be detected. This occurred when the conductor was on surfaces where the fault current was less than 20 A peak. The relay also failed to detect faults of short duration: 1 to 2 seconds. Arc noise signal coupled from a faulted feeder to another unfaulted feeder on the same bus; therefore, it was difficult for the relay to determine the faulted line. The presence of grounded–Y–connected capacitor banks on the feeder resulted in preventing the noise signal from travelling to the detector. The use of tuning inductors on the capacitor banks to allow the arcing fault signal to reach the substation was not effective for 1200 kVAR and larger capacitor banks.

- To overcome the poor propagation of the high frequency components of the fault current past a capacitor bank, the technique was extended to include off–harmonic signals near 60 Hz [10]. Arcing causes non–synchronous signals which can be differentiated readily from the synchronous 60 Hz and harmonics signals. While arcing faults produced only subtle changes in the fundamental current, they caused substantial amplitude changes in off–harmonic frequency components. Noise activity resulting from switching transients produced insignificant variations in the off–harmonic signals. Signal coupling to other feeders was expected to be greatly attenuated, in contrast to high frequency signals.

Band–pass filters with steep roll–off, or a switched–capacitor comb filter centered at 30, 90, 150, and 270 Hz was suggested to obtain the signals of interest. The filter would reject the 60 Hz and harmonics simultaneously.

The resulting signal would be further filtered to block direct current (dc) and eliminate high frequency components. The remaining signal would be a wide–band off–harmonic signal with a frequency range 10 to 300 Hz. The energy algorithm could then be applied to obtain the level of off–harmonic signal in the system.

- A detection algorithm [11], based on the random nature of arcing associated with HIF, is activated once the 2 to 6 kHz harmonic "energy" in a cycle deviates from an average by a certain predefined percentage. The number of transitions from arcing (high level of energy) to non–arcing (low level of energy) during a short period of time (30 cycles) was counted. During the same time, the number of adjacent cycles which have energies that differ from each other by more than a certain threshold was also counted (to detect faults with longer arcing bursts). If either counter exceeded its threshold number, a fault was signalled. This algorithm was also sensitive to the level of high frequency activity in the current signal.

- The ideas discussed above were integrated in [12]. The sampled current signal was digitally filtered to extract a variety of signals: the high frequency components (2 to 6 kHz); even, odd, and off–harmonics; and the positive, negative and zero sequence components at the fundamental power frequency (60 Hz) of the current. The summation of the square of the filtered current data samples, over an entire power frequency cycle, was calculated in a digital signal processor (DSP) chip to determine their "energy" level in that cycle. The detection algorithm was implemented on a microprocessor based system. This system provided time counters, lookup tables for weighting constants, and the logic to update thresholds dynamically. A knowledge–based

system (expert system) interacted with the microprocessor to combine algorithmic and heuristic approaches for problem identification and solution.

The data analysis of this system showed that the detection parameters were not only sensitive to the type of ground material but also to the moisture content in the soil. It was also suggested that the faulted phase could be identified as the high frequency component of the arc current occurred at certain position on the voltage cycle, i.e. when the system voltage is equal to the restrike voltage.

***Kwon et al** [13]*

Along the same line, Kwon et al relied on the half–cycle asymmetry in the high impedance fault current waveform to obtain a detection parameter. The FFT of a 64 sample–per–cycle of the neutral current signal was taken. The ratio between the "power" contained in the even harmonics (6^{th} to 32^{nd}) to the "power" contained in the odd harmonics (7^{th} to 33^{rd}) was calculated: note that for a sampling rate of 64 sample per cycle the highest harmonic available would be the 31^{st} [14]. The term "power" is similar to the term "energy" described earlier. This ratio was used as a detection parameter in an algorithm similar to the one mentioned above. Testing of this algorithm showed that its effectiveness dropped to 30 % as the fault current became less than 20 A.

***Ebron et al** [15]*

Ebron, Lubkeman, and White showed that the problem solution could be achieved much faster if the detection parameters were parallel processed by an artificial neural network (ANN) rather than by an expert system. A distribution feeder, capacitors, and induction motors fed from power electronic

circuit were simulated using the Electro Magnetic Transient Program (EMTP). The high impedance faults were simulated by switches connected to impedance elements (of the order of load impedance). Data generated by simulation were obtained by varying the size of motor loads, fault impedance, and switching times. Twenty parameters were computed for each 512 sample–per–cycle–per–phase window to represent the status of the system undergoing a transient. The magnitude of the fundamental, sequence and harmonic components, as well as the "energy" over ten frequency bands of the current were calculated, to extract the features of HIF and other switching transients occurring along the feeder. The results of the simulation are available in reference [16]. It is doubtful that the simulated transient waveforms represents an arcing fault or a typical load–switching transient.

The extracted parameters were input to an ANN of 200 input nodes (20 nodes x 10 cycles), 200 first hidden layer nodes, 400 second hidden layer nodes, and one output node. The network required 120,400 weights and 601 biases for full interconnection. The network learned from a 50–vector training set in 38 iterations using the back–propagation algorithm. It is not clear whether or not fault detection would require this network structure.

Hughes Aircraft Co. *[17]*

Hughes Aircraft Company designed a high impedance fault detector that used the third harmonic current as a fault indicator. On a Y–connected system, a fault was indicated if the relative phase–angle between one phase and the other two phases suddenly changed by at least 15° concurrent with an increase in single–phase fundamental current component of at least 15 A,

and this condition persisted for at least 5 seconds. Detector evaluation showed that it could trip under no–fault conditions.

Since phase relationship measurements on a Δ–connected circuit are quite difficult, three third harmonic magnitude monitors were built for Δ–connected circuits. A fault was indicated if a predetermined percentage increase in the third harmonic current level concurrent with a single phase increase in fundamental current of at least 15 A for at least 3 seconds. The detector did not experience false trips in its evaluation period. In the evaluation period of both relays there were no naturally occurring downed conductor faults.

Jeerings & Linders *[4][18][19][20]*

Based on their research on the nature of high impedance faults, Jeerings and Linders characterized HIF as being highly resistive and nonlinear; consequently, the low order harmonic currents generated at the fault will tend to peak coincident with the voltage peaks, regardless of current magnitude (the analysis was based on symmetrical waveforms). This characteristic is not associated with any other single system phenomenon.

The third harmonic component of the fault current was separated from the fundamental and other harmonics. The phasor value of the 3rd harmonic was measured using magnitude and phase–angle detectors. The phasor signal was averaged over a short and a long period of time. Short time averaging was used for present signal determination. Long time averaging determined the 3rd harmonic current level in the system. The fault harmonic current magnitude may be erratic; therefore the difference between these averages would contain information of any change in the third harmonic

current in the network. A fault was indicated when this change lies within a specific phasor window.

The third harmonic of the system voltage was processed similarly. It was suggested that the phasor ratio between the change in the third harmonic system voltage to the change in the third harmonic current, referred to as the sink impedance, would determine whether the fault occurred upstream or downstream from the point of measurement, and would add to the *security* (relay's ability to **not** trip when it shouldn't) of the system. However, even if the normal power system may contain some nominal level of third harmonic voltage, the harmonic voltage may not change. The measurement of the sink impedance will not yield any additional information over that gained from monitoring the third harmonic current only.

1.3. Summary of Fault Conditions and Detection Methods *[2]*

High impedance faults may not be associated with arcing. To give a complete picture of the problem, the possible electrical conditions of high impedance faults and fault detection methods are summarized in this section.

1.3.1. Electrical Conditions of High Impedance Faults

A downed conductor can result in eight basic electrical conditions that can be grouped as follows :

1. ***Broken Conductors***

   A broken conductor can result in four types of electrical conditions:

   i. no ground contact or a very high impedance to ground.

ii. ground contact on the load side of the break with no back feed.

iii. ground contact on the load side of the break with some loads beyond the break connected phase–to–phase.

iv. ground contact on the source side of the break.

2. ***Sagging or Fallen Conductors***

A broken pole or pole hardware could result in a sagging conductor. The line current continues to flow. The conductor could either:

i. have no ground contact or a very high impedance to ground,

ii. be in contact with a low impedance ground.

3. ***Contact by Foreign Object***

The line is fully operational. The possibilities are:

i. line contact by an insulated or a very high impedance object, such as a rubber–tired crane.

ii. low impedance ground contact, such as a tree.

1.3.2. Summary of Detection Methods

The detection schemes proposed in the last few years could be classified as follows:

1. ***Detection Based on 60 Hz Measurements***

This approach is based on measuring 60 Hz quantities on the source side of the fault location. Load and ground current levels, and sequence voltages and currents are the measured quantities. Proportional relaying is an example of this category.

2. ***Loss of Voltage***

   A broken conductor could be detected by monitoring the voltage on the load side of the break. Communication is required between the load side and the substation.

3. ***Detection Based on non–60 Hz Measurements***

   The noise and harmonics produced by arcing in downed conductor faults have been used as a signature in many algorithms such as the 2 to 6 kHz signal, off–harmonic signals near 60 Hz, third harmonic current, and even and odd harmonics.

4. ***Imposed Signals***

   A pulse echo could be used to identify the end of a conductor. A sudden change in the conductor's end location indicates a conductor break. Another approach injects a high frequency signal and measures the response of the line.

5. ***Fault Enhancement***

   The method is to turn an undetectable low current fault into a fault that is detectable by conventional overcurrent devices. One approach connects 2 to 3 metallic rods, perpendicularly oriented to the line, to the neutral conductor on each span. A falling conductor contact with one of these rods is highly probable. The resulting high current ground fault is easily detected.

Each detection method is limited to detect only some types of fault. This limitation can be understood from Table 1.1; therefore, a multi–technique high impedance fault detector is expected to be a logical approach to a successful high impedance fault relay.

Type of Fault	Detection Method				
	60 Hz Measure-ments	*Voltage Loss*	*Noise & Harmon-ics*	*Imposed Signals*	*Fault Enhanc-ement*
1. Broken Conductors					
no ground contact.		x		x	x
load side ground contact.		x		x	x
as above with ph–ph load.	x	x	x	x	x
source side ground contact.	x	x	x	x	x
2. Sagging Conductors					
no ground contact.					x
ground contact.	x		x	x	x
3. Foreign Contact					
very high impedance.	x		x	x	
low impedance.	x		x	x	

Table 1.1. Limitations of fault detection methods to the type of faults they detect. (x = detection is probable) [2].

1.4. Research Objectives

The research objectives are to find an alternative approach to the detection of high impedance arcing faults, to design an algorithm of the fault detector, and to write the software that could be practically implemented in an integrated relaying hardware. The designed detector should be able to distinguish between faults and other system loads that could mimic arcing faults.

The following chapters give details of the study conducted to achieve the research goal.

2 Characteristics of High Impedance Arcing Faults

To study the nature of high impedance faults, field tests and a series of high voltage laboratory tests were conducted for data collection of fault current and voltage signals.

2.1. Field Tests

2.1.1. Test Circuit

The University of Manitoba and Manitoba Hydro conducted staged fault tests in a field two miles east of Sperling, Manitoba, in September of 1989 [21]. The test arrangement is illustrated in Fig. 2.1. The source of power was the 7.5 MVA, 66/25 kV, 65/173 A transformer bank at Carman substation. The bank impedance is 6 %. The maximum load at the substation is 60 A. The fault level at the substation is 100 MVA for a three–phase fault. The corresponding fault current on the 25 kV side is 2300 A at a 100 MVA base, considering 0.2 per–unit system impedance. Substation protection is a three–phase oil circuit recloser (OCR). Recloser tripping is activated by phase overcurrent relays. The ground overcurrent relay is not in service. The phase–relaying is set at 200 A ± 10 %. On the high voltage side of the transformer, 125 A fuses are used for backup protection.

The fault site was 32 circuit kilometers east of Carman substation. The first 20 km of the circuit is a three–phase line. The line normal load current

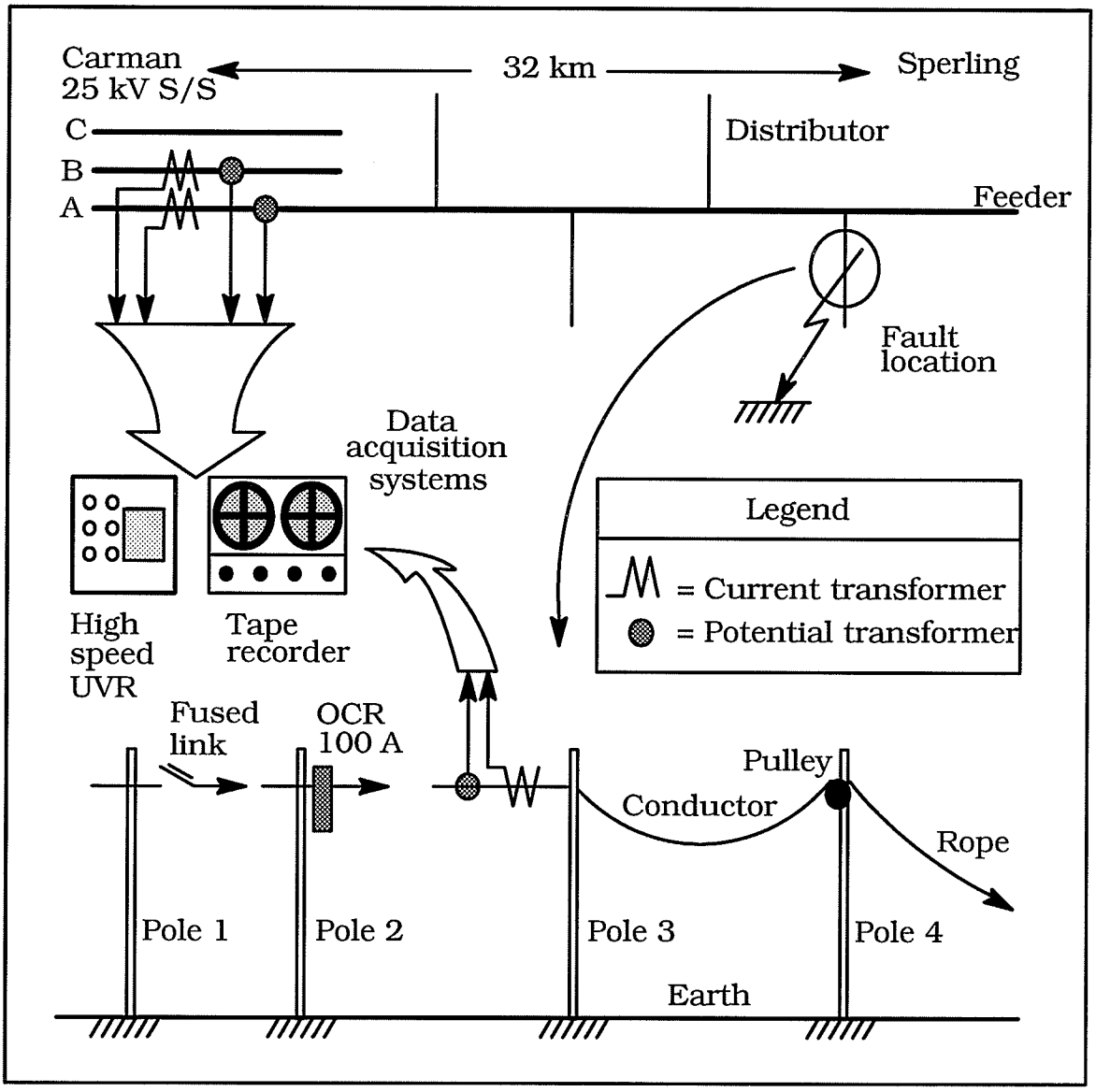

Figure 2.1. Staged high impedance fault field test arrangement.

is 30 A. The remaining part of the circuit is a single–phase single–conductor line. The normal load current on this portion of the circuit varies between 0 to 10 A. The short circuit level at fault location is 268 A. The line protection is a 140 A OCR 10 km east of Carman, and a 25 A fuse to protect the single–phase line. Test circuit data are shown in Fig. 2.2.

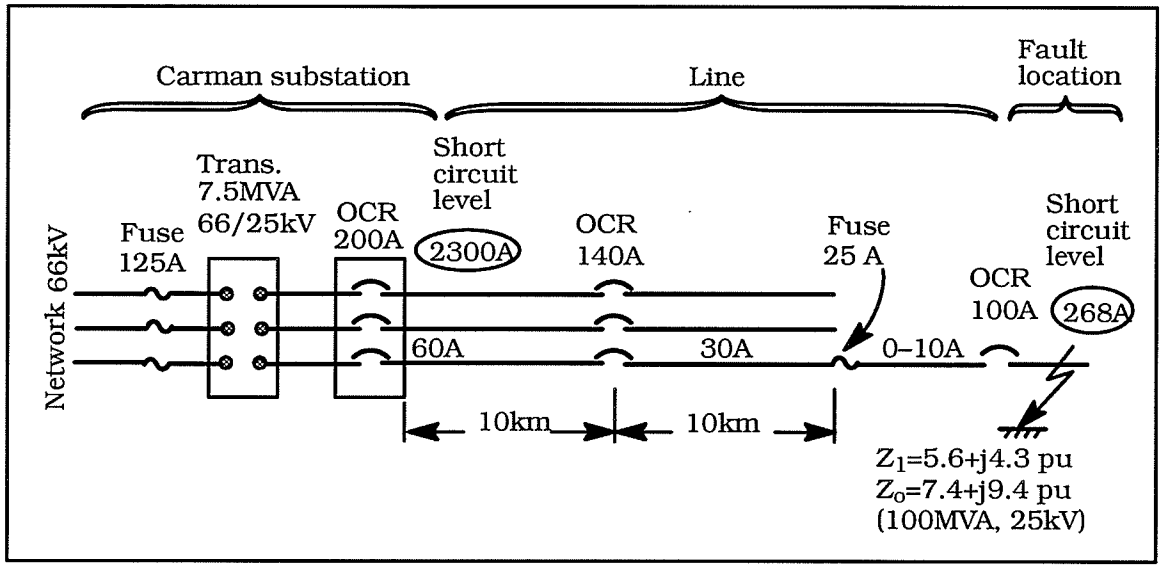

Figure 2.2. Test circuit data.

During staged fault tests, the trip level of the substation OCR was doubled to 400 A, the 140 A OCR and the 25 A fuse protection were disabled, and an OCR, set to trip at 100 A with one second delay, was installed at the fault location. Figure 2.3 is a photograph of the fault location, at Sperling, showing the conductor to be dropped to ground to initiate faults.

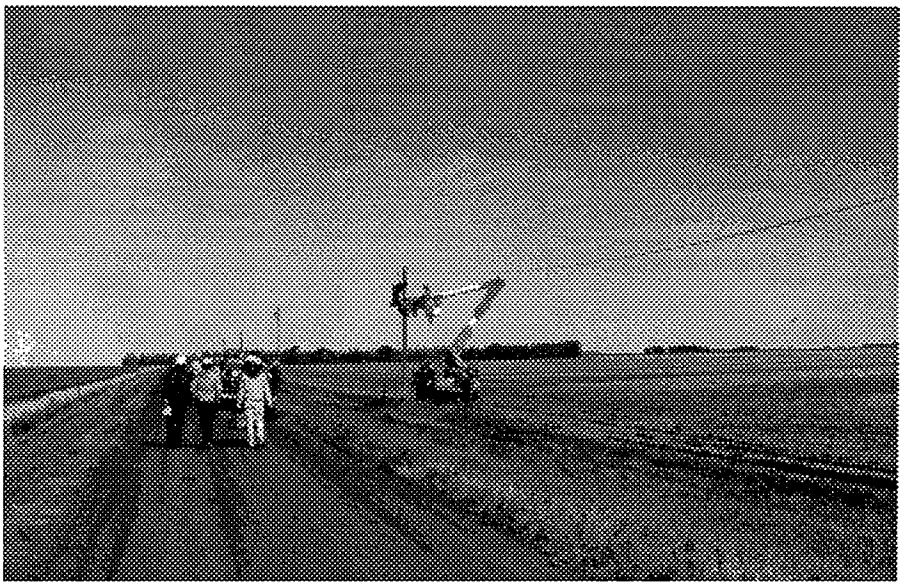

Figure 2.3. Site of the high impedance fault field tests.

2.1.2. Instrumentation

The instrumentation supplied by the University of Manitoba for data acquisition were tape recorders of band width dc (direct current) to 20 kHz, and oscilloscopes with output amplifiers. The oscilloscope input channels have a band width (BW) of dc to 20 MHz—input signal from a 50 Ω terminated source, input resistance of 1 MΩ, and input capacitance of 7.5 pF. The output channels BW is dc to 10 MHz, and produce 75 mV_{pp}/div from approximately 600 Ω source. The gain of the oscilloscope input amplifiers was adjusted to adapt the signal level input to the tape recorders. Manitoba Hydro used high speed ultra–violet recorders (UVR) to record the current and voltage waveforms during tests. Figure 2.4 shows a photograph of the University of Manitoba data acquisition system at Carman substation.

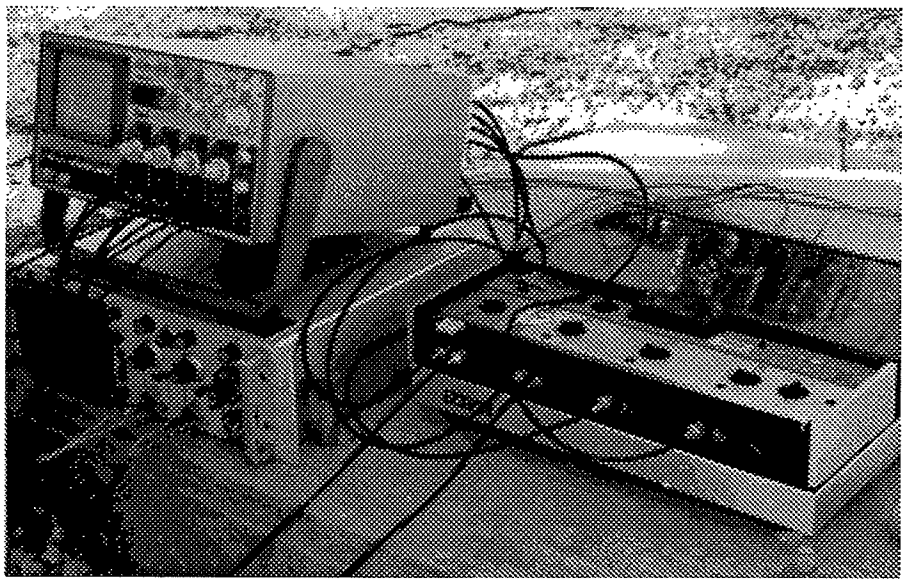

Figure 2.4. Instrumentation at Carman substation.

Currents at the substation were sensed using the 200/5 A current transformers (ct's) enclosed in the three–phase OCR. The voltage signals were obtained from the 14,400/120 V substation service transformers. At the fault

location a 300/5 A current transformer, BW dc to 50 kHz, and a 14,400/120 V potential transformer (pt), BW dc to 6 kHz, were used as current and voltage transducers respectively. The BW of these transformers were determined in the laboratory using a signal generator driving a power amplifier. The amplifier capability was limited by its transformer. These BWs could be exaggerated; however, the BW of interest in data acquisition was dc to 1 kHz.

The burdens of the ct's were 120 mV/A current–to–voltage transformers, BW dc to 10 kHz, shown in Fig. 2.5. Voltage sensors of ratio 25:1 (40 mV/V transformers) were used with the pt's: BW dc to 15 kHz.

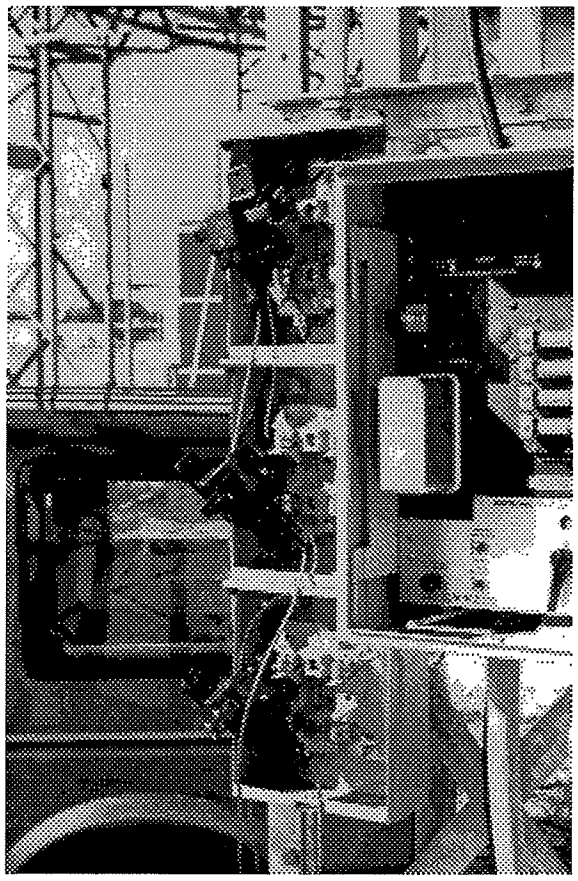

Figure 2.5. Current sensors used as burdens of current transformers.

2.1.3. Observations

The current and voltage waveforms of a fault on the field soil are shown in Fig. 2.6. The photograph is taken off a dc to 500 kHz storage scope. The measured voltage is the voltage between the high voltage conductor and an earth rod driven at the fault location (pole 3 in Fig. 2.1). The impedance at the point of fault was not high enough to limit the fault current to the unprotected region by the 100 A OCR at the fault site. Fault current record length averaged between 6 to 12 cycles. The current waveforms shown in Fig. 2.6 had an initial value of 55 A peak. The current gradually increased in magnitude to 165 A peak after three cycles. Two cycles thereafter, the OCR tripped. It was noticed that the fault voltage decreased and became flat topped as the current magnitude increased; the fault current waveform became nearly sinusoidal. The phase relationship was mainly resistive in contrast with that of short circuit faults, where the arc is ignited in a largely inductive circuit.

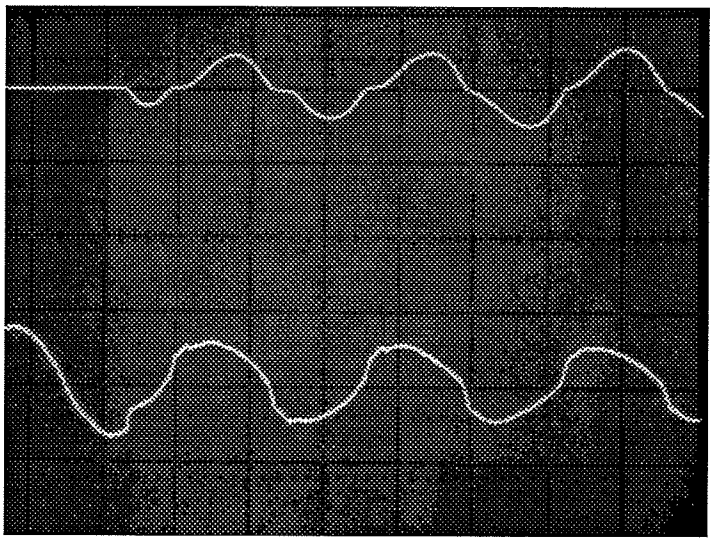

Figure 2.6. High impedance arcing fault current and voltage waveforms.
Top: current 275 A/div. Bottom: voltage 25 kV/div.

In another test, the energized conductor was dropped on a wooden pallet, Fig. 1.2, the arc elongated and caused a 17 cm burn on the wood. This could be due to magnetic forces (motor action) tending to move the arc away from the source [22].

Considering the substation and line normal loads, and the fact that ground protection is not active, a downed power line on the test circuit is extremely dangerous; the fault current could vary between 0 and 200 A at medium distribution voltage [1]. This raises a question upon the degree of safety, regarding HIF, at all similar locations of Manitoba Hydro's network.

2.2. Laboratory Tests

2.2.1. Setup

A high voltage laboratory setup, Fig. 2.7, was devised at the University of Manitoba for further investigations and data collection of high impedance faults under controlled conditions. The 25 kVA, 240/7200 V, 105/3.5 A, 1.5 % impedance, distribution transformer used in laboratory tests was energized from a 100 A, 208 V, 60 Hz supply via a starting resistor. A bare conductor—one end connected to the high voltage side of the transformer and the other to an insulated rope—was dropped to ground to initiate a fault. The ground was a pile of soil placed on a metal pan. The ground return was completed through a variable current limiting resistor. The current and voltage signals were delivered to a data acquisition system from a 300/5 A current transformer, and a 10 000/10 V high voltage probe: BW dc to 1 MHz at 20 kV peak–to–peak. A photograph of the laboratory setup is shown in Fig. 2.8. Arcing associated with HIF resulted in energy dissipation in the

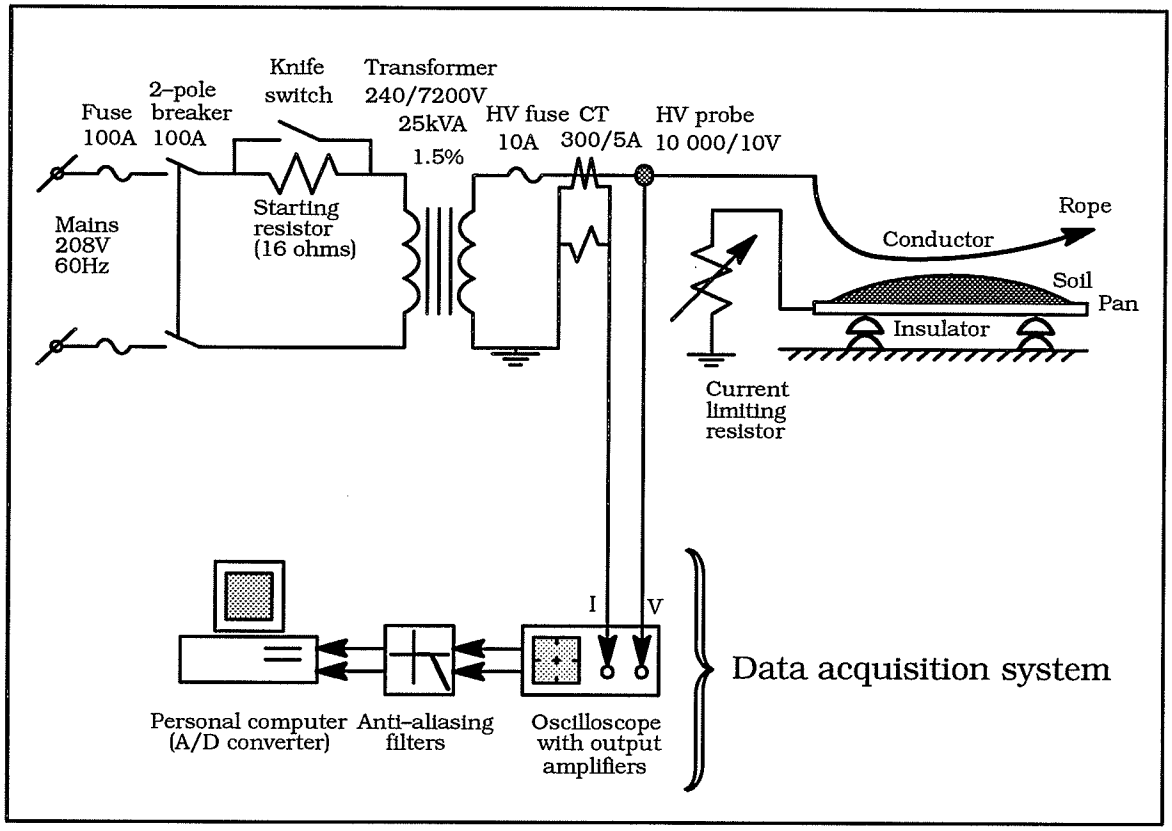

Figure 2.7. Circuit diagram of staged high impedance fault laboratory tests.

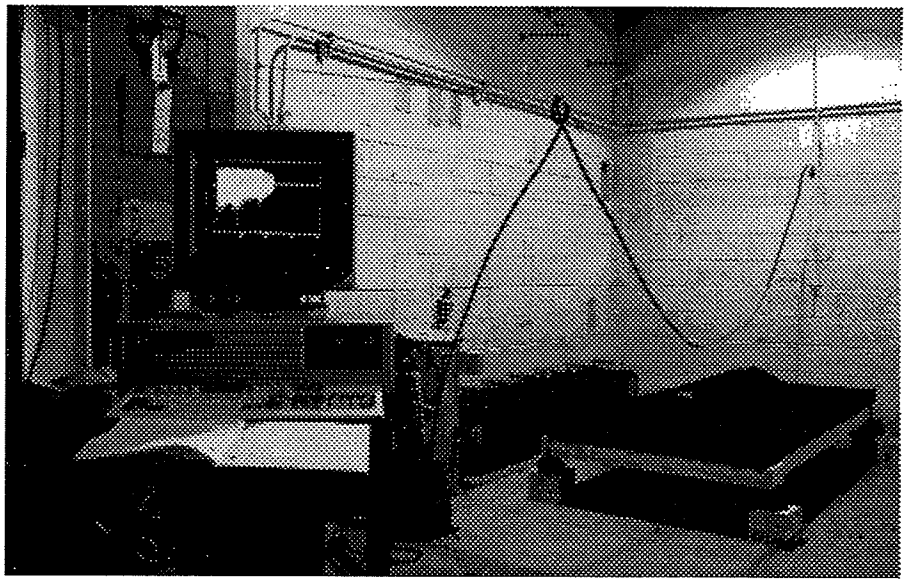

Figure 2.8. Photograph of the high voltage laboratory setup.

form of heat that turned the moisture in the soil into steam and burned the grass into smoke: Fig. 2.9.

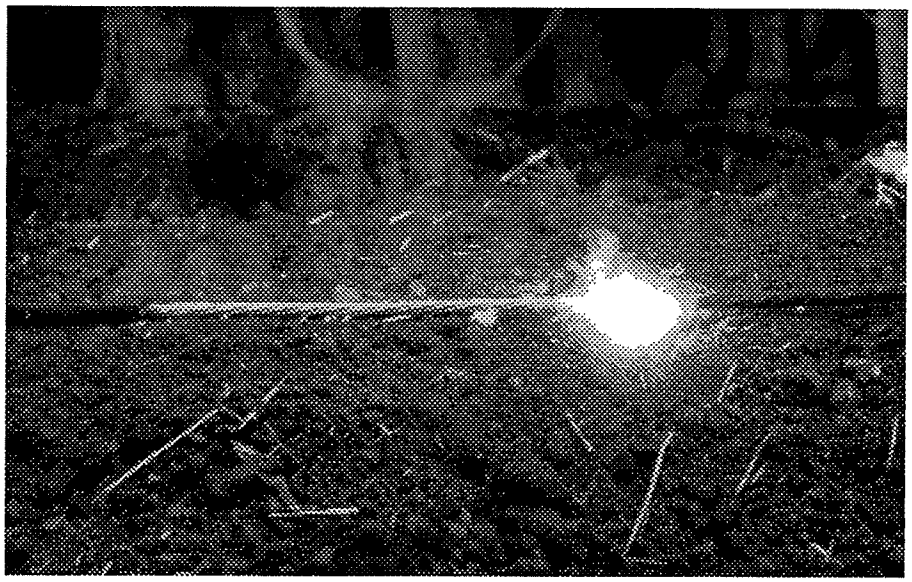

Figure 2.9. Arcing associated with HIF.

2.2.2. Data Acquisition System

The data acquisition system was composed of an analog–to–digital converter (A/D), anti–aliasing filters (AAF), and an oscilloscope with output amplifiers: see Fig. 2.7.

The A/D converter board, installed in a personal computer, is controlled and activated through *Lotus–123*™*. The board was set to sample at a rate of 32 samples per cycle, i.e. a sampling frequency of 1920 Hz, which is the rate applicable to many modern practical microprocessor based relays. The data were quantized to ± 2048 bits. Board input impedance is 1 GΩ. The current and voltage signals were collected simultaneously and stored in a worksheet. The personal computer was able to store 4.3 seconds of data per–run at the required sampling rate.

---

* *Lotus–123*™ is a trademark of Lotus Development Corporation

Resistor–capacitor (RC) anti–aliasing filters, R=47 kΩ and C=0.0033 μF, were used to prevent folding of high frequency signals around the Nyquist frequency as low frequency signals. The filters were set at 1.0 kHz cut–off frequency and 20 dB/decade attenuation. The input impedance of the A/D board, and the source impedance of the oscilloscope output channels have a negligible effect on the filter characteristic. The oscilloscope adapted different transducers to the selected input signal level of the A/D converter board, provided an on–line display of the measured signals, and isolated the computer from any unexpected hazard that could occur in the test circuit.

2.2.3. Observations

A photograph of a typical fault current waveform obtained in a laboratory test is shown in Fig. 2.10. At the passage of current through zero, an off–current conduction period is observed for approximately 2 ms until the voltage magnitude becomes large enough to break down the small air gaps between the conductor and earth, and initiate arcs.

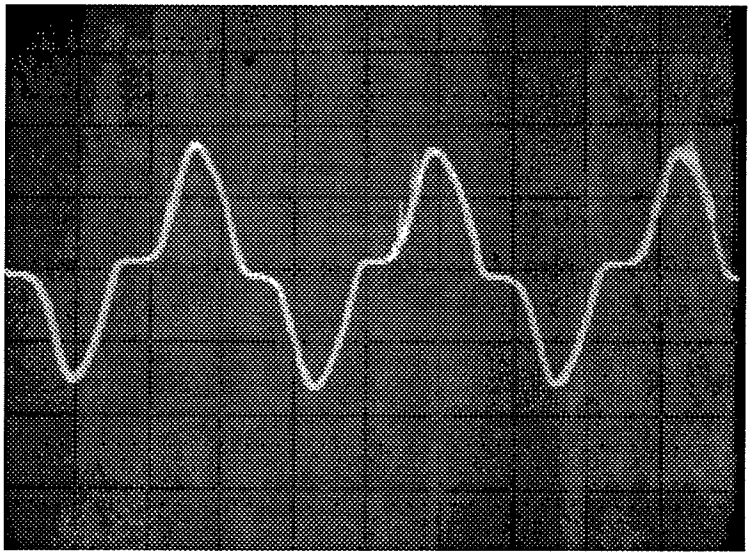

Figure 2.10. Laboratory test fault current waveform: 2.5 A/div.

2.3. Voltage – Current Characteristic of High Impedance Arcing Faults [2][4][7][22][23][24]

The voltage–current (v–i) characteristic of an arc is entirely different from that of a solid conductor. Whereas the v–i characteristic across a conductor is linear (the voltage across the conductor is proportional to the current through it), the current flow in an arc has a different mechanism. Moreover, the arcing phenomenon associated with downed power lines deviates from that with conductor–to–conductor faults, or across circuit breaker poles. Compared to conductor–to–conductor faults, arcing in high impedance faults occurs in a largely resistive circuit. It is characterized by short arc length and small current magnitude. HIF could persist for a long period of time resulting in a random arc behavior. The v–i characteristic of the high impedance arcing fault of Fig. 2.6 is shown in Fig. 2.11.

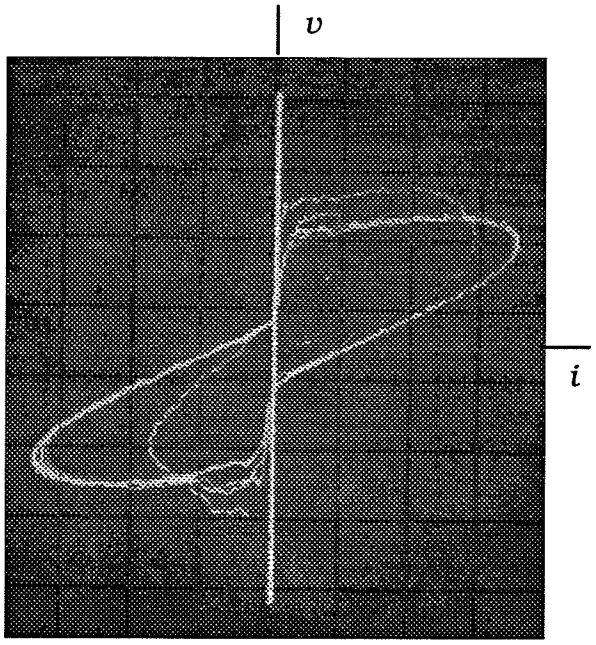

Figure 2.11. Voltage (v) – current (i) characteristic of an arcing fault.

2.3.1. Arcing Mechanism

Consider an energized power line falling to the earth surface below. At the initial contact of the conductor with earth, Fig. 2.12(a), the voltage gradient at the *conductor–soil interface* is large. Local ionization results. The gap conducts as the voltage across it reaches the breakdown voltage V_b at current zero and time T_b: Fig. 2.13. The small contact area, via the arc tip in Fig. 2.12(b), accounts for the observed reduced initial current of HIF. The conductance of the air decreases as the current of the established arc increases. The voltage drops to V_{arc}: Fig. 2.13.

A thermionic emission process starts to build up as the electric field between the electrodes, conductor and earth, emits electrons from the cathode spot. The liberated electrons ionize neutral molecules electrically. These ions heat up the electrodes as they fly towards both of them under the electric field strength effect. Typical values of temperature at the arcing spot are about: 2000 to 3000 °C for metallic electrodes; 3000 to 4000 °C for carbon electrodes; and 5000 to 8000 °C in the gas column [24]. The conductive layer of the soil moves away from the conductor. The arc penetrates the earth between the soil particles enlarging the effective contact between the conductor and the ground: Fig. 2.12(c). The increase in the effective contact area between the conductor and the earth is a source of nonlinearity in arcing HIF. It is also accounted as the source of high impedance in these faults. The current reaches its maximum, at time T_m, as the applied voltage becomes equal to the arc voltage: Fig. 2.13. The current then starts to decrease and returns to zero as the balance between the rate of heat generation of arcing, and the heat transferred to the environment is disturbed. The volt-

age between the electrodes of a burning arc (V_{arc}), Fig. 2.13, drops with this decreasing current. The arc is extinguished at time T_e. At this stage either the moisture will defuse back into the dry soil and the arc will be re–ignited,

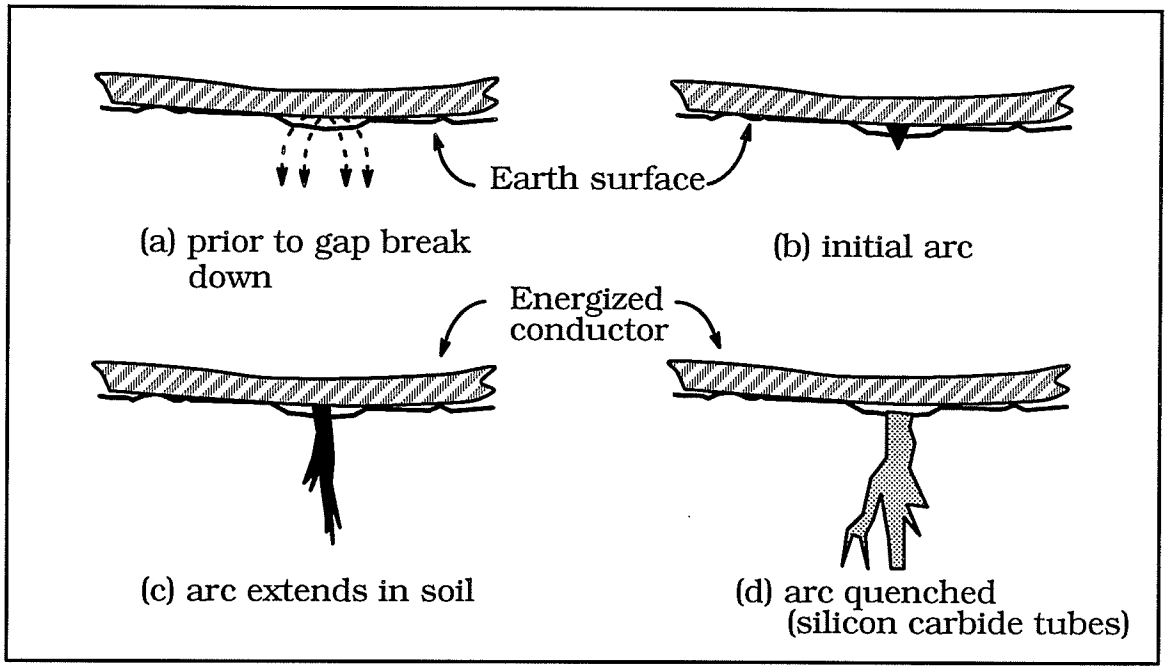

Figure 2.12. Arc formation in a downed power line [23].

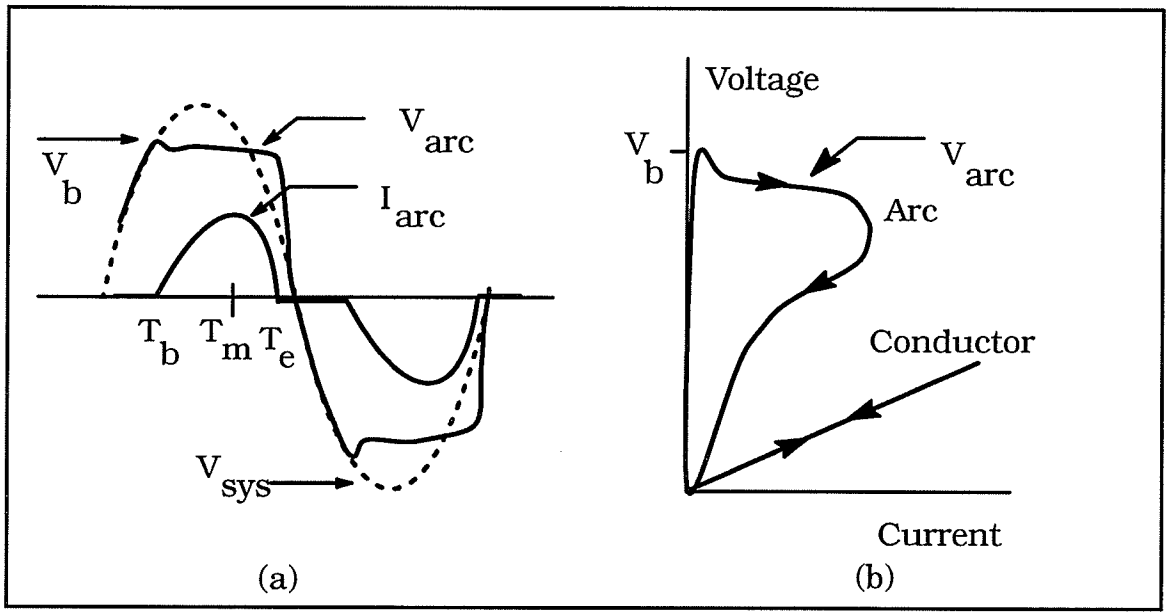

Figure 2.13. Physics of ac arc: (a) waveforms. (b) v–i characteristic.

on a different path than the previous, once the front edge of the moist (conductive) layer is close enough to the high voltage electrode. Or, another point which was previously inactive may start arcing if the local field has increased due to changes in the potential distribution.

The resulting voltage–current characteristic of an arcing HIF, therefore, consists of a branch for increasing current and another for decreasing current: Figures 2.11 and 2.13. This arc hysteresis is due to the heat capacity of the conductor, earth, and arc gas. The temperature of arc column and electrodes, and thus the arc voltage, correspond still to preceding conditions of the current rather than following the instantaneous conditions. Because of the poor heat conduction of the earth soil, the two branches of the characteristic shown in Fig. 2.11 are widely different.

The arc heat is enough to fuse sand and silica in the soil into a glass–like substance, silicon carbide: Fig. 2.12(d). These glass–like tubes, shown in Fig. 2.14, reach a length of 5 cm. They were found to have a linear resistance of the order of 2 to 100 kΩ/m [23].

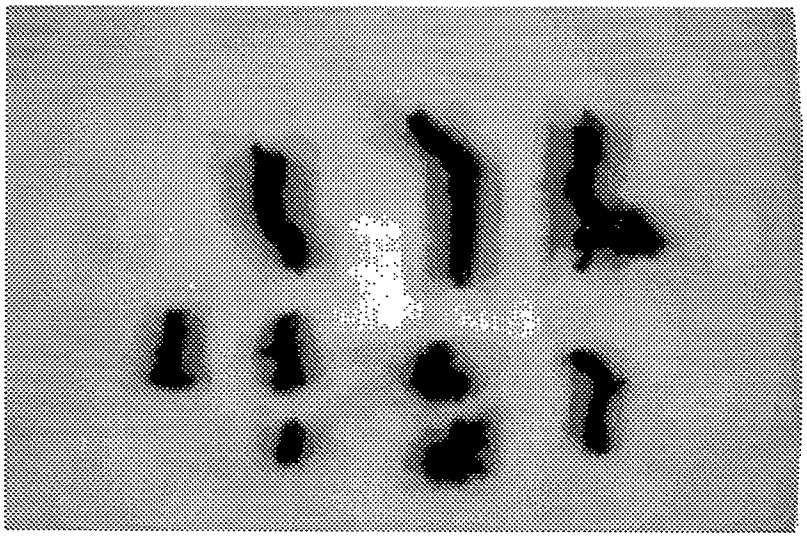

Figure 2.14. Arc fuses sand and silica into glass–like tubes.

The nonlinearity of the arc and conductor–soil interface, the development of silicon carbide tubes, the generation of smoke and steam, the bounce of the conductor on the ground surface, the movement of the soil particles, the moisture content in the soil, and the ground material itself interact in a complex arrangement to produce the overall v–i characteristic.

Faults on Dry and Wet Soil

The shape of the characteristic changes with the arc geometry and the heat transfer conditions. Both elements are affected by the moisture content of the soil. Faults on drier soil require higher ignition voltage to ignite the arc. The effective air gap length is larger than that of faults on wet soil; therefore, the current waveform would be more distorted for faults on dry soil compared with an ordinary sine curve. The higher the arc voltage becomes the more distorted the arc current is; the interval through which the current stays at zero, as shown in Fig. 2.10, is longer.

On wetter soil, the conductor does almost have a solid contact with the soil. The fault current starts sinusoidal. As the heat evaporates the moisture at the soil surface, air gaps are created at the conductor–ground interface and arcs are triggered. The arc length is short and the arc voltage is small. As time goes on, the steam increases the soil porosity, the arc starts to propagate in the soil, the arc voltage increases, and the current waveform becomes distorted.

2.3.2. *Fault Current Asymmetry and Randomness Behavior* *[2][23]*

It has been observed that the magnitude of the fault current may vary greatly from one cycle to the other, and that the positive half–cycles of the

current may be greater in magnitude than the negative half-cycles, or vice versa. Figure 2.15 illustrates these phenomena.

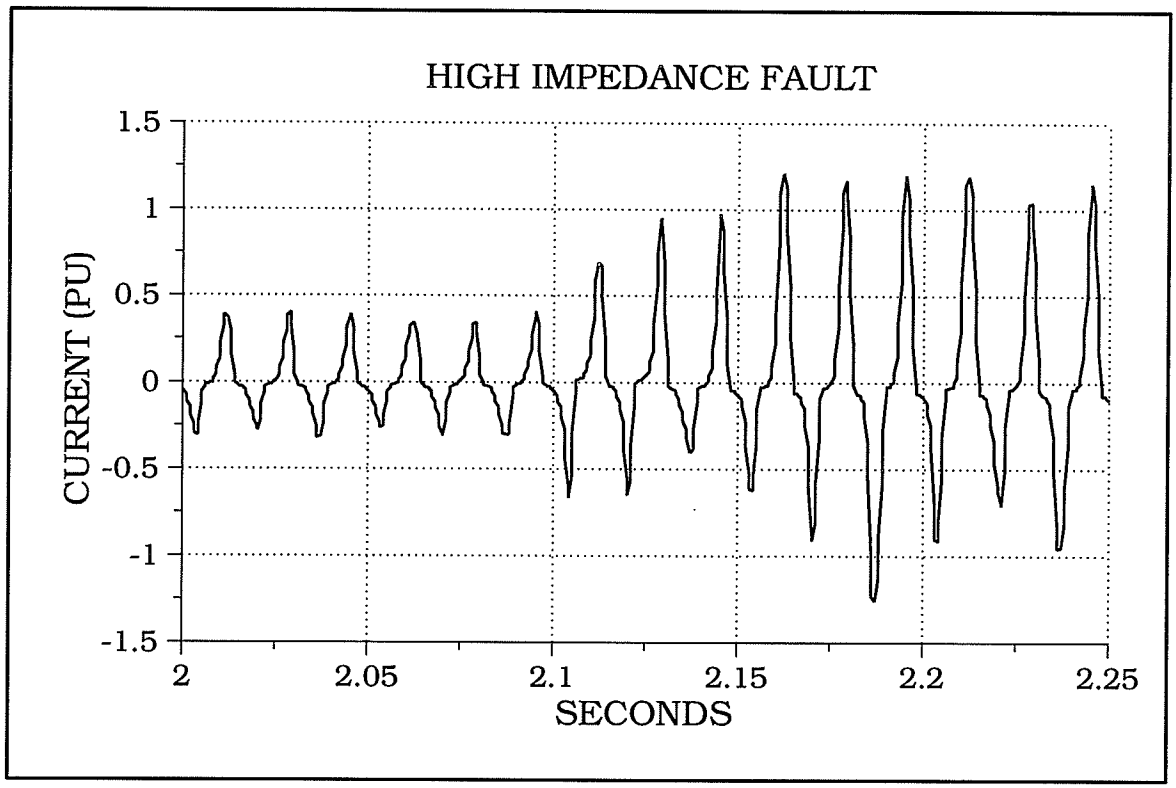

Figure 2.15. Asymmetry and randomness in HIF current waveform.

Fault current asymmetry and randomness could be explained as a result of arcing at the fault. The heat produced by arcing converts the moisture in the soil into steam. The steam expands and displaces soil which rearranges the characteristics of the air gaps surrounding the downed conductor. As a result, the current in the next arcing cycle could be quite different from that in the previous one. Furthermore, the silicon carbide tubes accumulate around the conductor; in the long run, the conductor could be insulated from the ground. The fault current magnitude, therefore, could reduce and make the situation worse. However, the impurity in the fused material could

on the contrary provide a solid path for the current, and the fault current would increase instead.

The asymmetry of the fault current in some cases could be as a result of the rectifying action exhibited by the soil. The glass–like tubes surrounding the conductor act as a hot cathodic spot that emits electrons. The voltage drop across the cathode spots is small when the conductor is positive: Fig. 2.16. The amount of moisture in the soil and the packing of its particles affect the values of the break down (onset) voltage of the gaps between the energized conductor and the earth. Less densely packed (drier) soil yields higher onset voltages and, therefore, a larger degree of asymmetry. Even order harmonics are generated on account of this asymmetry.

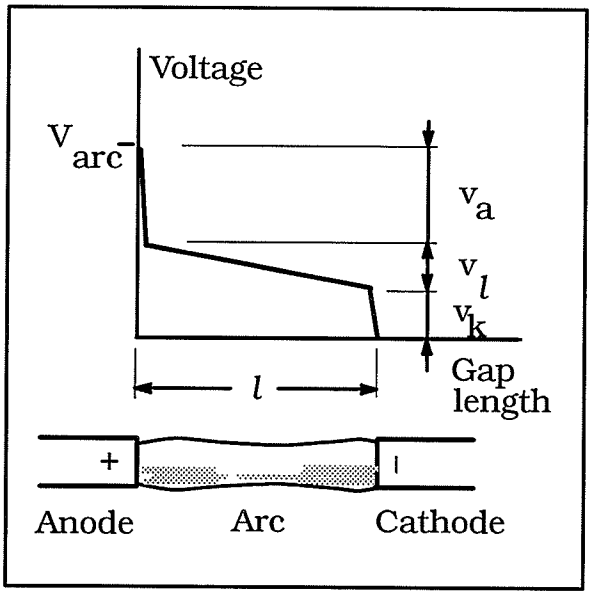

Figure 2.16. Voltage distribution between arc electrodes. v_a: anode drop, v_k: cathode drop, v_l: arc drop [24].

2.3.3. Faults on Snow Covered Ground

High voltage lines could be downed in the winter time due to ice loading on the conductors. The conductor and its supports could experience a me-

chanical stress beyond their design limits. Staged HIF tests were conducted in the high voltage laboratory to investigate downed power lines on a snow covered ground.

The soil was subjected to the winter climate during the period of December to February, where the temperature falls below –20 °C. The mobile tray was brought to the laboratory and a downed conductor fault was staged promptly. The energized conductor lay on the ground showing neither visual nor measurable indication of arcing. The energized conductor looked quite harmless: Fig. 2.17. This indicates how downed conductor faults on snow covered surfaces could be extremely dangerous.

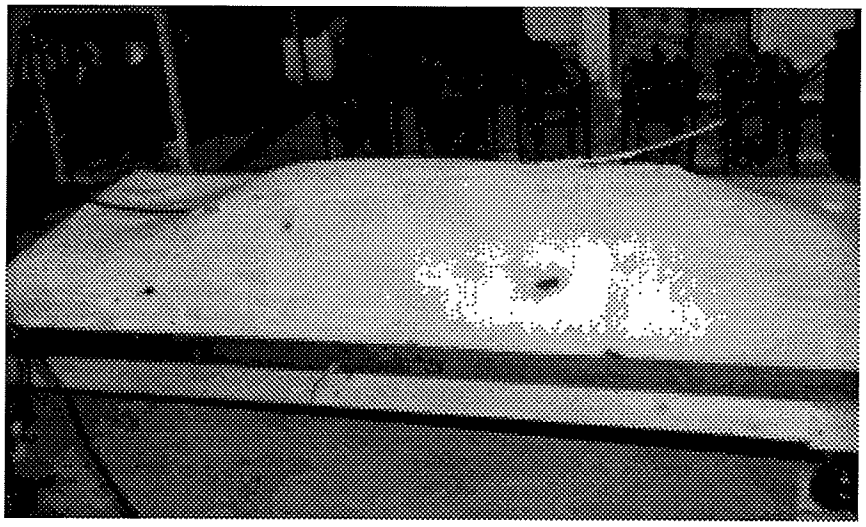

Figure 2.17. Fault on a snow covered ground.

The cold temperature produced a frozen layer at the surface of the soil; together with the snow cover, a perfect insulator was formed on top of the soil capable of insulating a 7200 V conductor. Even if the soil was not frozen, but the snow cover was thick or densely packed, the phenomenon should still be expected. This situation could last for a long period of time before the spring comes. Or (if lucky), a break occurs in the frozen layer, due to the

weight of the fallen conductor, and small arcs start to heat the contact area and puncture the insulation layer. It may take a while before the amount of fault current becomes significant.

2.4. Conclusion

The results obtained from the high voltage laboratory downed power line model were in agreement not only with the field test results but also with data published in the literature by other institutions and research groups. The power limitation of the test transformer forced the use of current limiting resistor in the fault circuit. However, the arc model was still justified. This validated the laboratory model, and established a credible source of data acquisition to launch further investigations toward solving the high impedance arcing faults problem.

3 Security Testing of High Impedance Fault Detectors

The primary motivation for high impedance fault detection is to improve safety rather than protecting equipment or enhancing system performance. The action taken to improve safety may be the same as that used in overcurrent detection: de–energize the affected portion of the circuit. Sensitive fault detection can impact the number and duration of outages on a feeder. One aspect of utilities' design goals in a high impedance fault detector is correct operation; especially, the detector should not falsely indicate the presence of a fault when there is none. A utility would prefer a detector failing to trip for some high impedance faults than accepting nuisance trips [2].

In this chapter the susceptance of existing high impedance faults detection algorithms to operate under no–fault conditions is investigated [25]. The objective of this study is to give a newer insight into improving high impedance fault relaying.

3.1. Introduction

High impedance faults (HIF) are a persistent problem on power system distribution lines because the current may be too small to be detected by conventional means.

Several high impedance fault detection schemes, mostly involving microprocessor algorithms, have been proposed or implemented. Many detectors

have been tested for *dependability* (ability to trip when they should), but few have been tested for *security* (ability to **not** trip when they shouldn't). The basis for all of the algorithm designs is some kind of waveform discrimination, involving for example the Fast Fourier Transform (FFT), or a combination of these interacting with a knowledge–based environment.

It was anticipated that loads such as *arc welders, computers, and fluorescent lamps* would share some characteristic features with HIF. These loads are referred to here as "high impedance fault–like loads" (HIFLL).

A variety of such common load waveforms was collected using a computer–based data acquisition system. The *security* of two existing algorithms [13][20] was examined for high impedance faults on dry and wet soil, arc welding machine, computer, and fluorescent lighting loads.

3.2. Data Collection

3.2.1. Laboratory Tests

High impedance fault data were collected using the model HIF setup shown in Fig. 2.7. Tests on dry soil were conducted with the current limiting resistance shorted. In wet soil tests, a 660 Ω resistor was inserted in the fault circuit to limit the fault current to the circuit rating.

The high impedance fault–like loads considered were:

1. ***Arc Welding Machine***, rated 45.5 A, 230 V_{LL}, 225 A arc current, $\frac{1}{8}$" steel rod type 6013.
2. ***Computer***, 5.0 A, 120 V_{LN}.
3. ***Fluorescent Light***, 15.0 A, 208 V_{LL}.

Current and voltage signals of HIFLL were sensed using a 1000/1 clip–on ammeter, BW 10 Hz to 50 kHz, and a 10/1 voltage probe, BW dc to 10 MHz, connecting the instrument transformer secondary circuits to the data acquisition system shown in Fig. 2.7.

3.2.2. Waveforms

The per–unit values of the actual phase current and voltage signals were calculated from the collected data using conversion formulas. The conversion factors depend on the base values of the apparatus used and the dc offset of the oscilloscope. The current and voltage base values of a load are the peak rated current and the peak rated phase voltage of that load. Table 3.1 gives a list of the base values used in each case.

Case study	Base current (A)	Base voltage (V)
HIF	3.5 √2	7200 √2
Arc welder	45.5 √2	132 √2
Computer	5.0 √2	120 √2
Fluorescent	15.0 √2	120 √2

Table 3.1. Current and voltage base values.

The nature of high impedance faults and fault–like loads could be understood from the voltage and current waveforms, Fig. 3.1, and the corresponding long–time current traces shown in Fig. 3.2.

The behavior of "high impedance fault current", Figures 3.1 and 3.2, is affected by the surface conditions. A fault on dry soil is characterized by unsymmetrical half–cycles, short current flow interval per half cycle, and a large degree of randomness. This is in contrast to a fault on wet soil. The

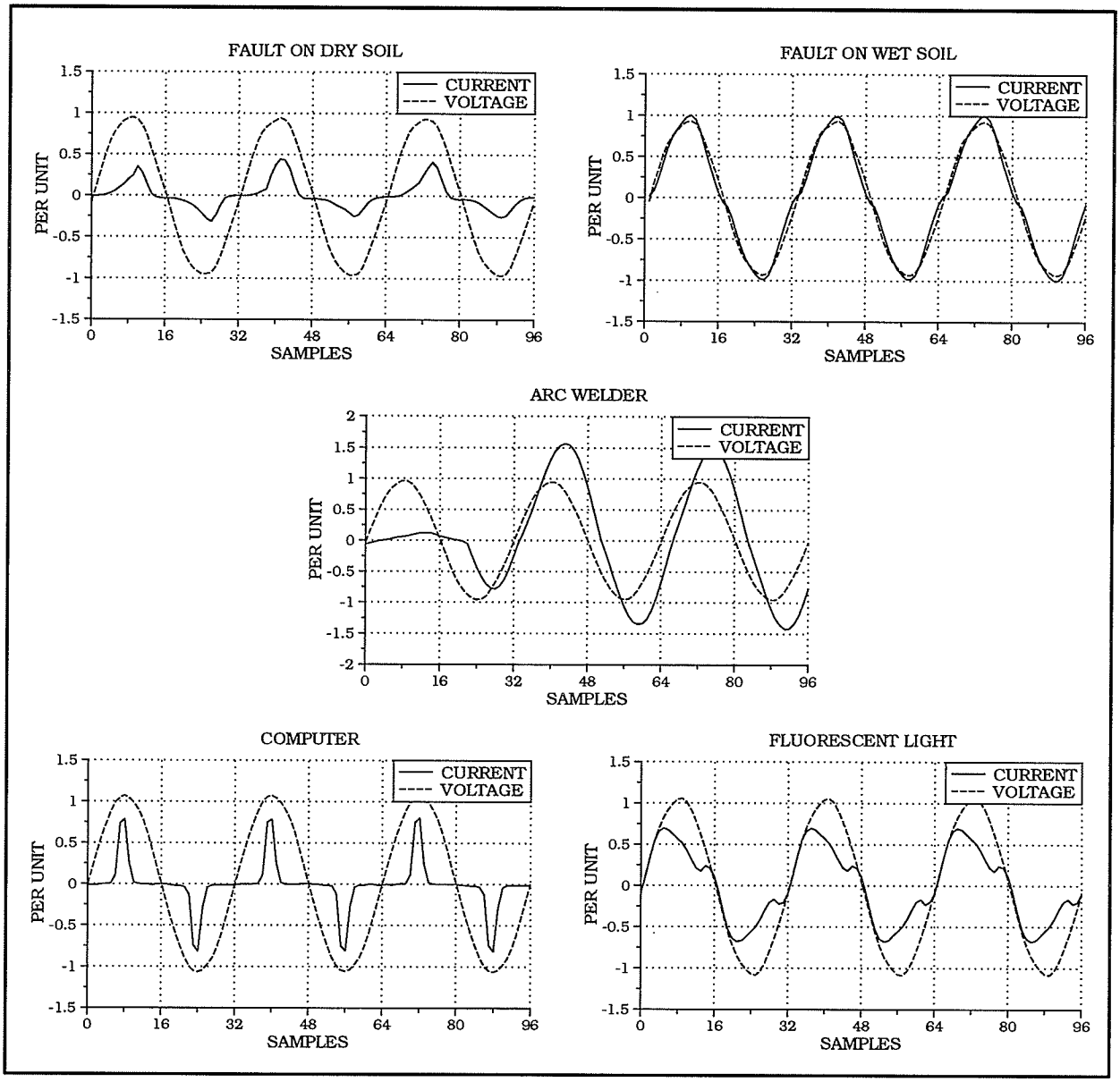

Figure 3.1. Current and voltage waveforms of HIF and HIFLL.

degree of dryness or wetness and the surface conditions would result in a different combination of these features for a given ground material.

The "arc welding machine current" is composed of two components: namely, the welding transformer magnetizing current and the arc current.

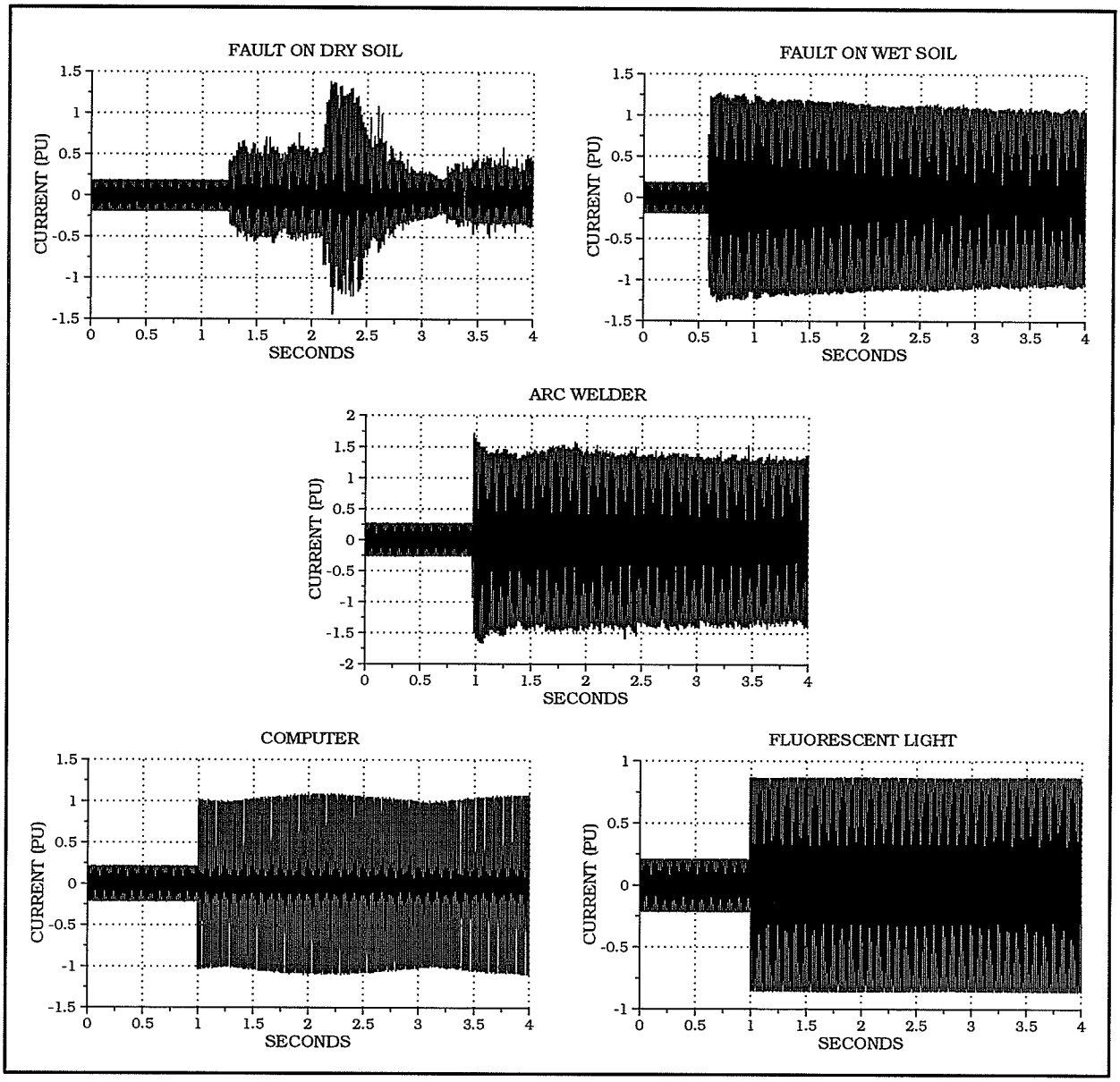

Figure 3.2. Current traces of HIF and HIFLL.

The arc current level modifies the waveform of the total welder current. The curve shape of the arc current is similar to that of an arcing fault except that the arc is burning in an inductive circuit. Short arc length is a characteristic common to both an arc welder and a HIF. Compared to the fault on wet soil, Fig. 3.1, the arc welder has a comparable current flow period per half-cycle.

The "computer current", Fig. 3.1, shows similarity to that of fault on dry soil. However, the narrower and symmetrical peaks in each half–cycle are distinctive. The periodic variation in the current trace is due to the cooling fan in the computer unit.

The "fluorescent lighting load" represents a widely used non–linear load. Fluorescent tubes follow the same rules of arcing as in an inductive circuit [24]. In order to avoid excessive flicker, they should not show any marked interruption of current as seen in Fig. 3.1. Waveform distortion suggests that this load contains harmonics used in HIF detection.

3.3. Data Processing

The HIF detection algorithms to be examined were the third harmonic phasor algorithm, and the even–to–odd harmonics power algorithm.

3.3.1. Third Harmonic Phasor Algorithm *[20]*

The third harmonic phasor algorithm responds to the *vectorial change* in the third harmonic current. The value (*Value*) of the third harmonic current is calculated each power–frequency cycle. The phasor difference between a short and a long time exponential average *(Avg)*, of 0.1 and 0.9 time constants "τ" respectively, equation (3.1), is determined. The relay operates when the detection signal lies within a predefined phasor window. The current magnitude window is set depending on the state of the system. The phase–angle window limits are 170 to 335 third harmonic degrees, lagging the 60 Hz substation voltage.

$$Avg_{new} = Avg_{old} * \tau + Value_{new} * (1 - \tau) \qquad (3.1)$$

3.3.2. Even–to–Odd Harmonics Power Algorithm [13]

The even–to–odd harmonics power algorithm calculates the ratio R_{e-o} between the sum of "powers" P contained in the 6^{th}, 8^{th}, ... , 32^{nd} harmonic to the sum of "powers" contained in the 7^{th}, 9^{th}, ... , 33^{rd} harmonic, each power frequency cycle: equation (3.2). The term "power" is used as an indication of arcing rather than an absolute measure of power. The detection threshold level is 1.5.

$$R_{e-o} = \frac{\sum_{e=6,8}^{32} P_e}{\sum_{o=7,9}^{33} P_o} \tag{3.2}$$

The "power" contained in a harmonic per–cycle was calculated by squaring the harmonic magnitude. Considering the sampling rate of 32 samples per–cycle, equation (3.2) was modified to equation (3.3); the highest harmonic available is the 15^{th}. Scaling factors of "100" are added for data processing reasons. To avoid calculation errors, the ratio R_{e-o} was set to "0" if either the nominator or the denominator of equation (3.3) is less than "1".

$$R_{e-o} = \frac{\sum_{e=6,8}^{14} (I_e * 100)^2}{\sum_{o=7,9}^{15} (I_o * 100)^2} \tag{3.3}$$

The necessary data processing required the Fast Fourier Transform taken for the current and the voltage signals each cycle. Appendix B lists the computer program, in *MathCAD*™* software and notation, used in testing

* *MathCAD*™ is a trademark of MathSoft, Inc.

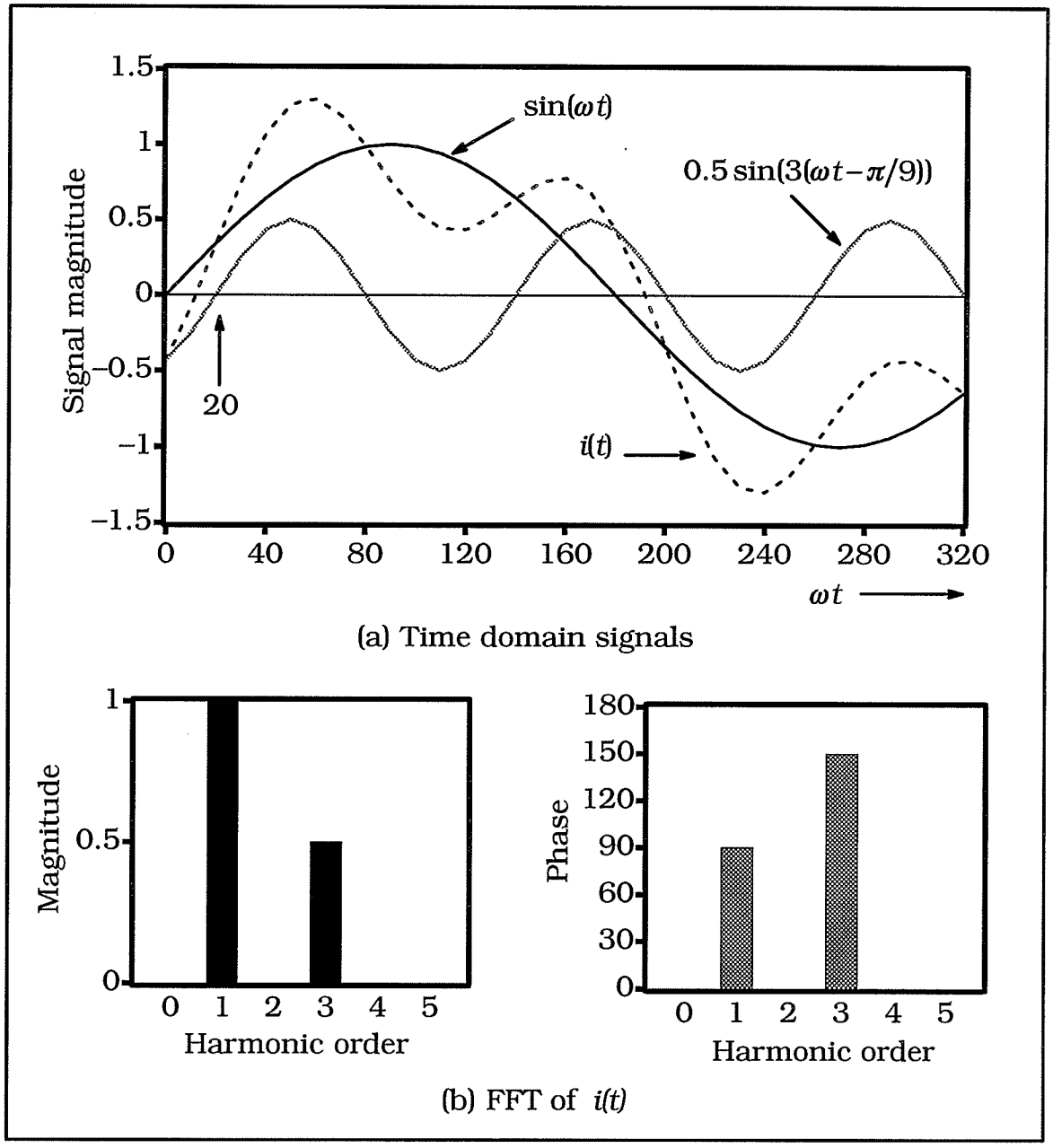

Figure 3.3. Time and frequency domain of a signal.

the security of HIF detectors. The resulting vector of coefficients c for the FFT of an n element real data vector v is [26]

$$c_j = \frac{1}{\sqrt{n}} \sum_{k=1}^{n} v_k \; e^{2\pi i(j/n)k} \tag{3.4}$$

where i is the imaginary unit, j and k are the indexes of the vector of coefficients and the data vector respectively. Figure 3.3 is an example illustrating the time and frequency domain of the signal $i(t) = sin(\omega t) + 0.5sin(3(\omega t - \pi/9))$. In the frequency domain, the FFT uses each harmonic as the 360° base when calculating the phase displacement.

The use of *RC* anti–aliasing filters, in the laboratory tests, produced a phase–shift depending on the harmonic order. The amount of phase–shift compensation for the 3rd harmonic current with respect to the fundamental harmonic of the voltage is + 6.77° at 1 kHz filter cut–off frequency.

The FFT was calculated for the current waveforms of Fig. 3.1. The frequency spectrum of the high impedance faults, and the fault–like loads are shown in Figures 3.4 and 3.5 respectively. It is clear that the distorted current waveforms result in spectra rich in harmonics. Because of the changing magnitude of the current waveform of fault on dry soil, the FFT of the three cycles shown in Fig. 3.1 was calculated; the results are illustrated in Fig. 3.6.

The arc welder, Fig. 3.4, is difficult to discriminate from arcing faults. The odd harmonics of fault on wet soil, computer and fluorescent light loads, Fig. 3.5, are dominating the even harmonics. This is because of current half–cycle symmetry. The Fourier analysis shows for fault on dry soil, Fig. 3.6, that the frequency spectrum varies from one cycle to the other. This variation accompanies a phenomenon of a random nature.

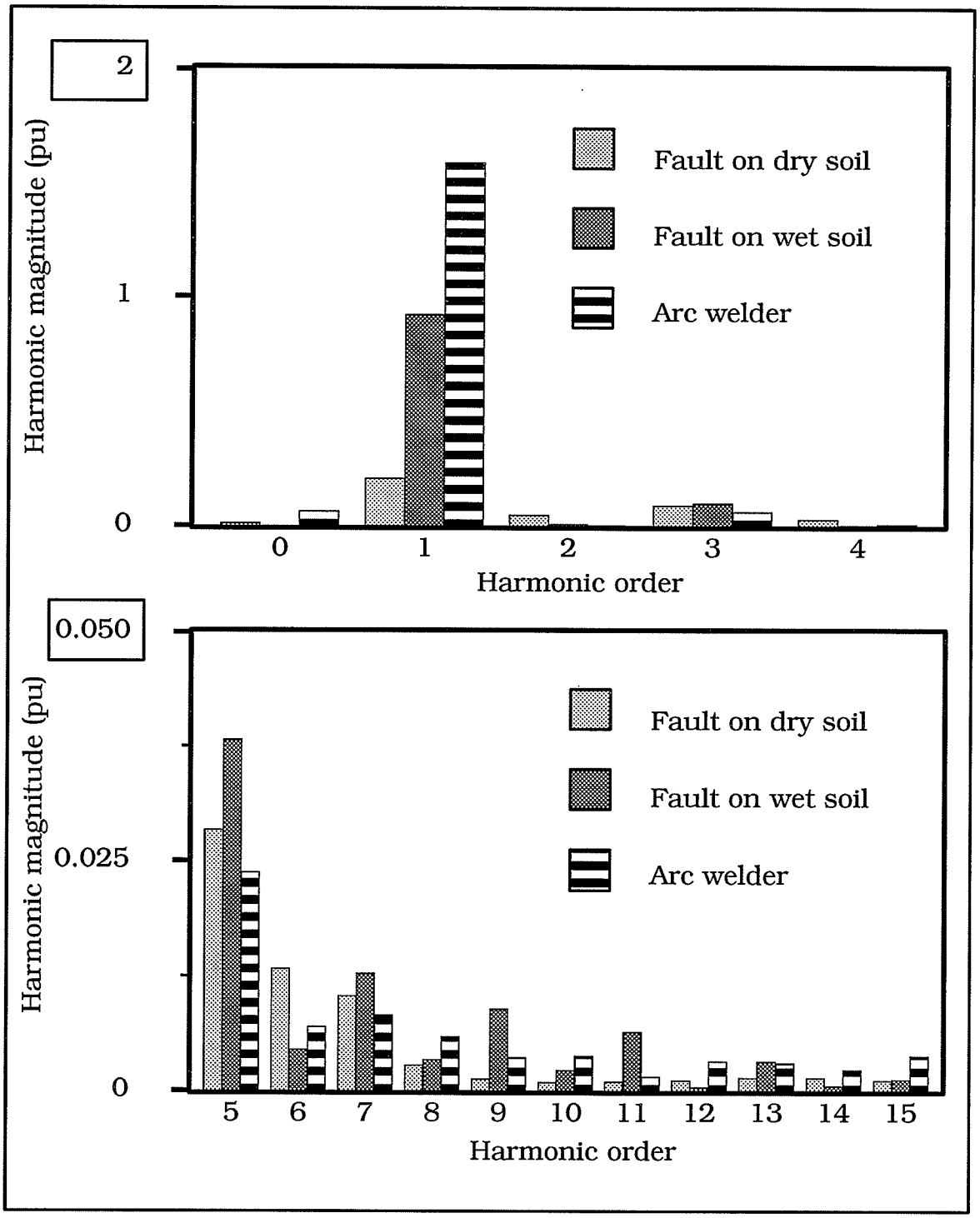

Figure 3.4. Frequency spectra of the HIF and the arc welder.

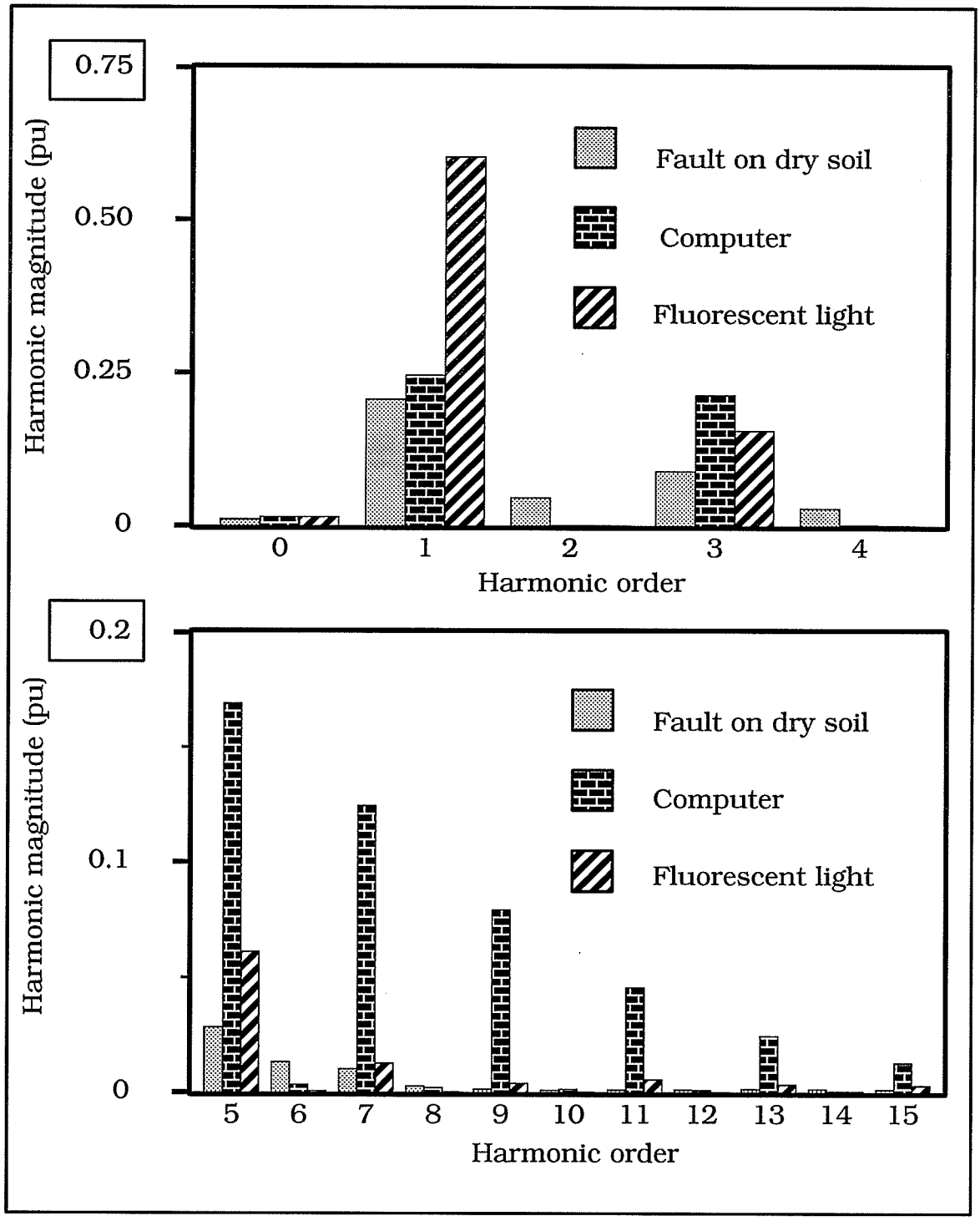

Figure 3.5. Frequency spectra of the computer and the fluorescent light compared to that of the fault on dry soil.

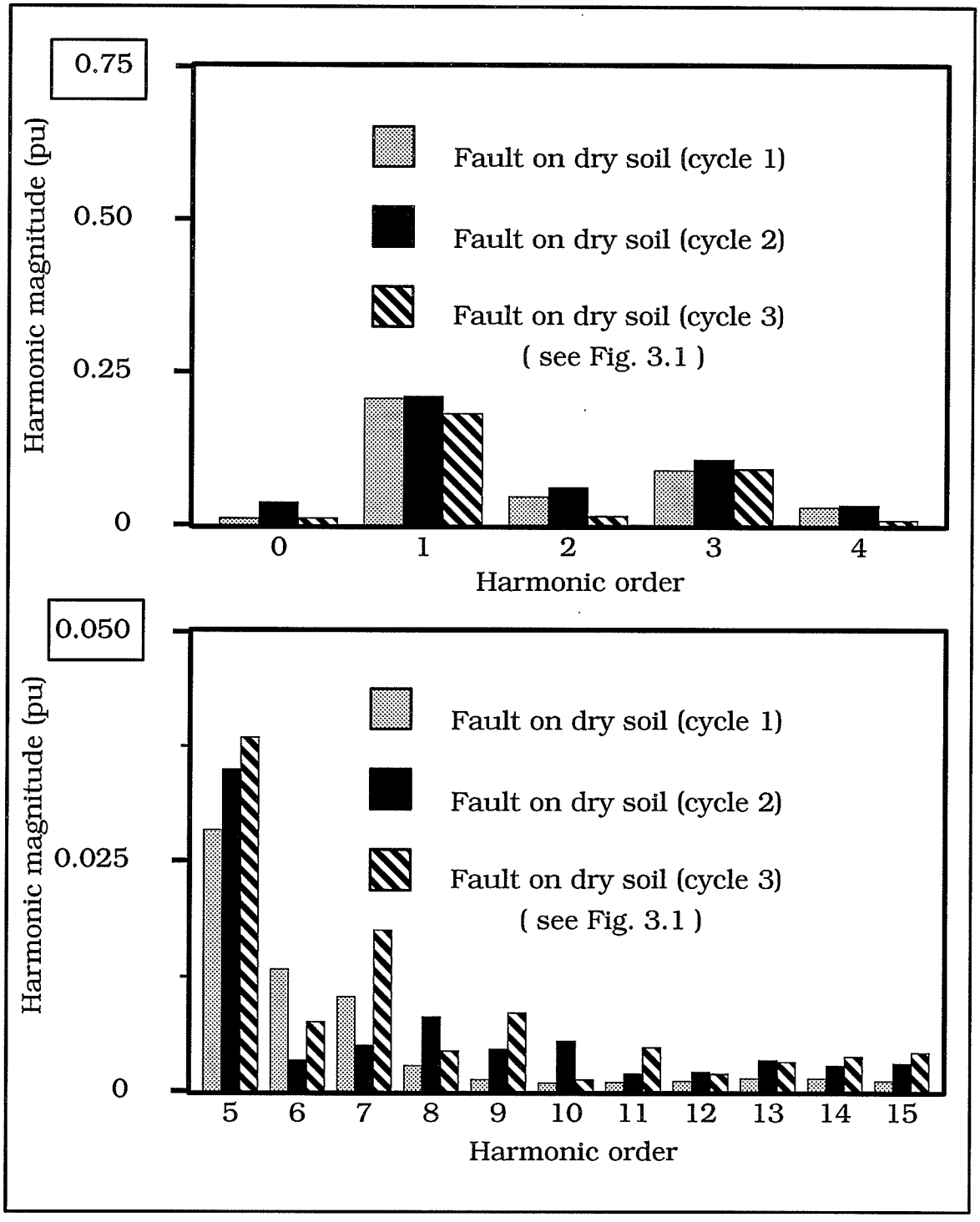

Figure 3.6. Frequency spectrum of the fault on dry soil in three successive cycles.

3.4. Results

The current traces illustrated in Fig. 3.2 were those on which the algorithms were tested. Each trace is composed of a 20 % artificial resistive load plus the fault or the fault–like load current. The artificial resistive load signal was obtained by multiplying the voltage samples by 0.2.

3.4.1. Third Harmonic Phasor Algorithm

The *change* in the 3rd harmonic current magnitude and phase is shown in Figures 3.7 and 3.8 respectively. In the first few cycles of normal load, a lack of 3rd harmonic current change is observed; the change in 3rd harmonic current component of the arc welder transformer magnetizing current shrank to a negligible value within half a second.

A sudden change is noticed at the instant of fault or fault–like load application. Third harmonic current exists in all cases. The use of the exponential averaging provides adequate follow up of the randomness in the waveforms as well as the trends in the 3rd harmonic ambient. It has also emphasized that the randomness exhibited by the arc welder is more than that of the fault on wet soil. These results support the observations on the waveforms shown in Figures 3.1 and 3.2.

In contrast to the fault on wet soil, the random burst nature of the arc on dry soil results in a noticeable variation in the 3rd harmonic current magnitude change. Once the arc current remains at almost a constant level, the magnitude change becomes in the same order of magnitude as in the remaining cases. The ambient 3rd harmonic magnitude is in the order of 1 to 2 % of the base value of fault or fault–like load current. Unless the fault has a high degree of randomness, the 3rd harmonic magnitude change is

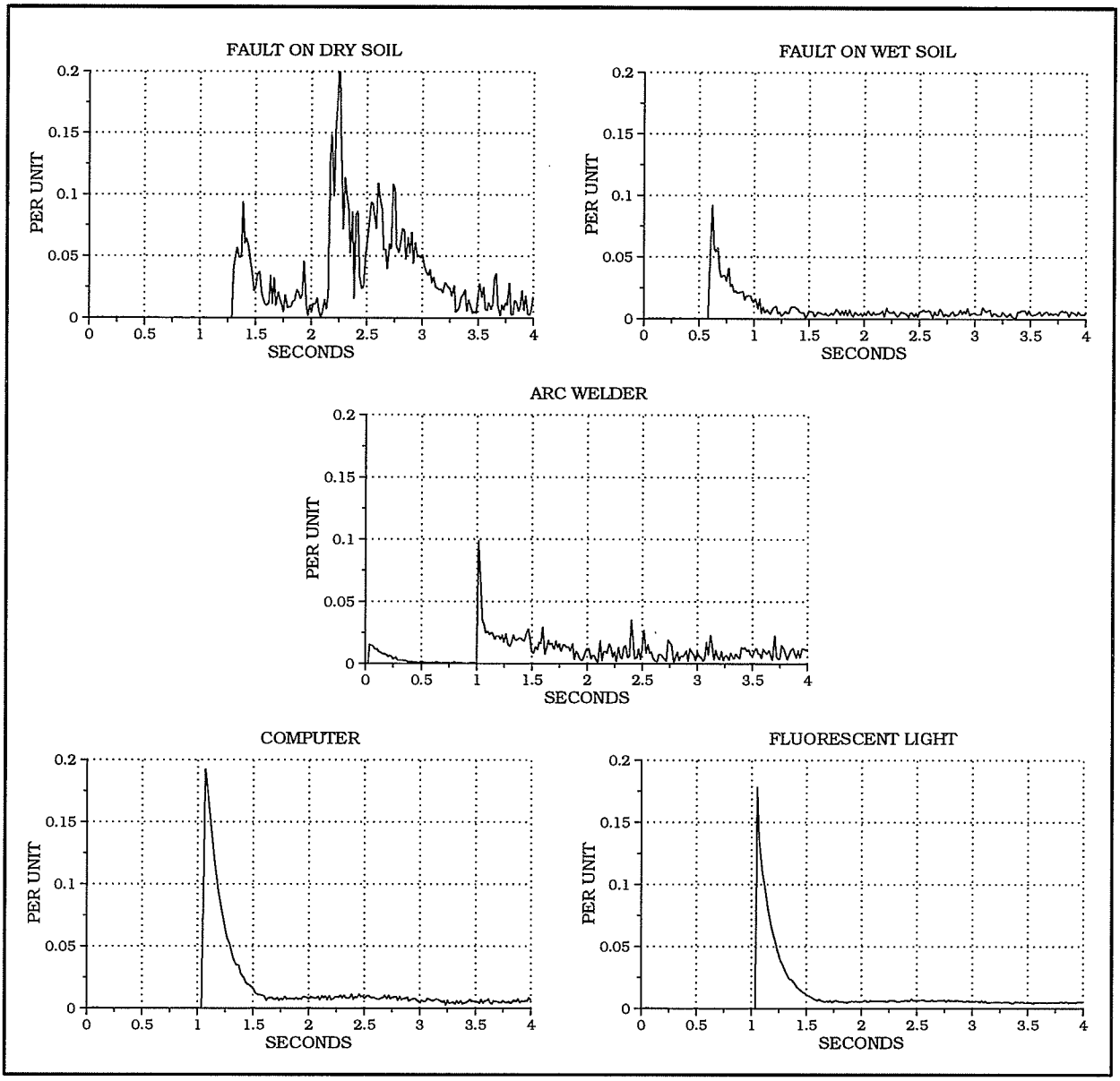

Figure 3.7. Magnitude of the 3rd harmonic current change.

insignificant; therefore, the use of this quantity as a detection parameter seems to be unreliable.

The phase* of the 3rd harmonic current change, Fig. 3.8, varies erratically in the HIF and arc welder events. For fault on dry soil, the phase starts

*Positive degrees are lagging the reference 60 Hz voltage, and negative degrees are leading.

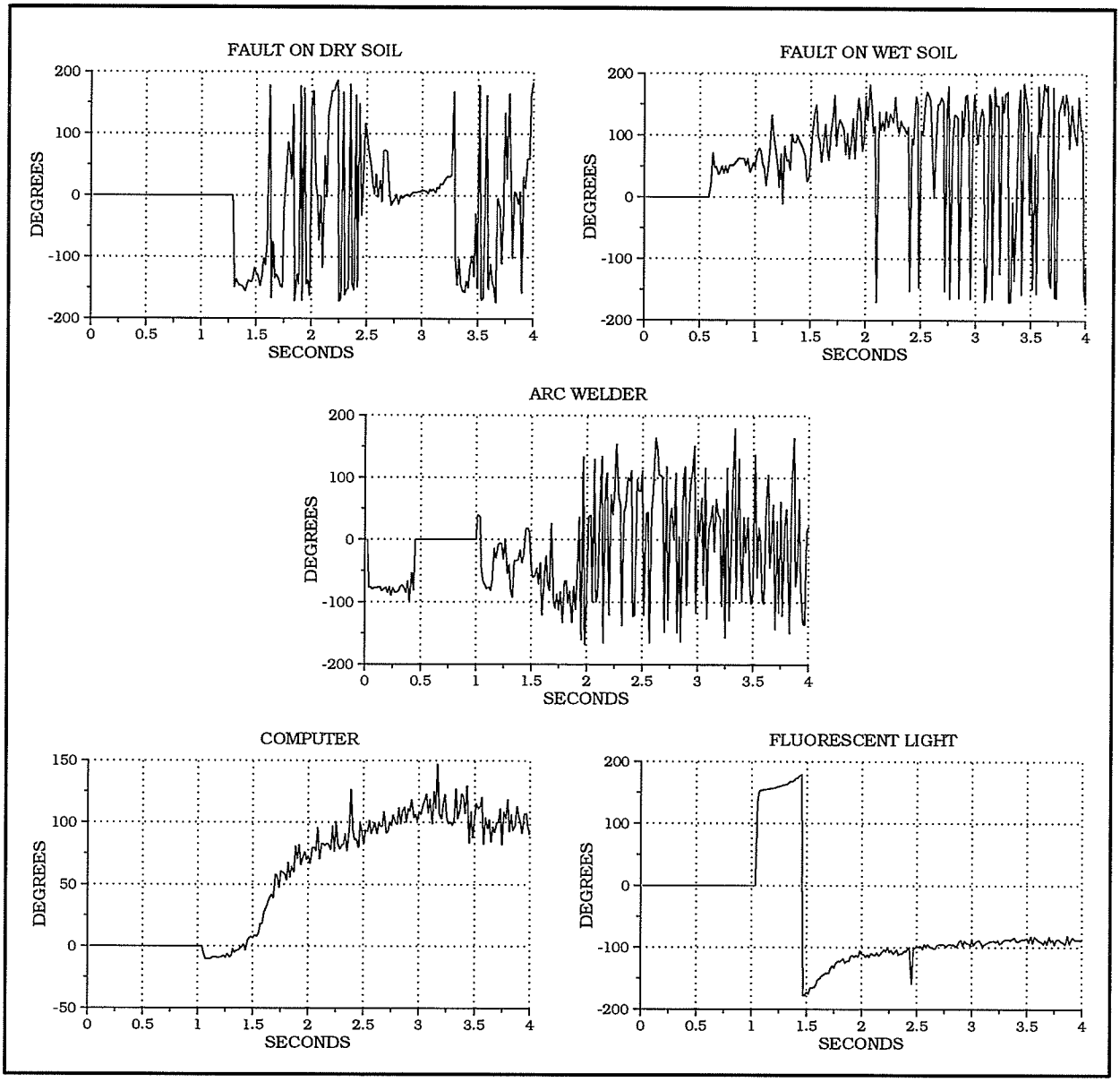

Figure 3.8. Phase of the 3rd harmonic current change.

at –150°, i.e. 150° leading or 210° lagging the 60 Hz voltage. A few cycles later the phase begins to vary widely between 150° lag and 150° lead, stays around 0° for half a second, then back again to its erratic variation. A similar description of phase behavior could be given to the fault on wet soil and the arc welder, where the phase variation lies within the same window limits.

The phase–angle trend of the computer and the fluorescent light loads is clear. It is reaching an average value of 100° lag for the computer and 100° lead, i.e. 260° lag for the fluorescent light. The phase angle window suggested in [20] would result in relay alarm in all cases except for the computer load. In the long run, it is expected that the change in 3rd harmonic current in the fluorescent light case would diminish. This would still leave the arc welder as a load likely to be confused with high impedance faults.

3.4.2. Even–to–Odd Harmonics Power Algorithm

The results of testing the even–to–odd harmonics power algorithm are shown in Fig. 3.9. The results are in agreement with the early discussion of the Fourier Analysis of the current waveforms. A similarity is noticed between the fault on dry soil and the arc welder. Another similarity is observed between the fault on wet soil, the computer, and the fluorescent light loads.

Since the number of harmonics used in the testing algorithm were less than that in the original algorithm, judgement on the performance of this detection method could not be perfectly justified. Nevertheless, the detection threshold could be lowered to 0.75, from 1.5 [13], to compensate for the missing harmonics. The results indicate that it is doubtful for a relay based on this algorithm to not signal a fault for an arc welder load. It is certain that a fault on wet soil will not be detected. For a fault on dry soil, if the degree of half–cycle asymmetry is not enough to generate a suitable amount of even–to–odd harmonics, the fault will not be sensed using this detection criterion.

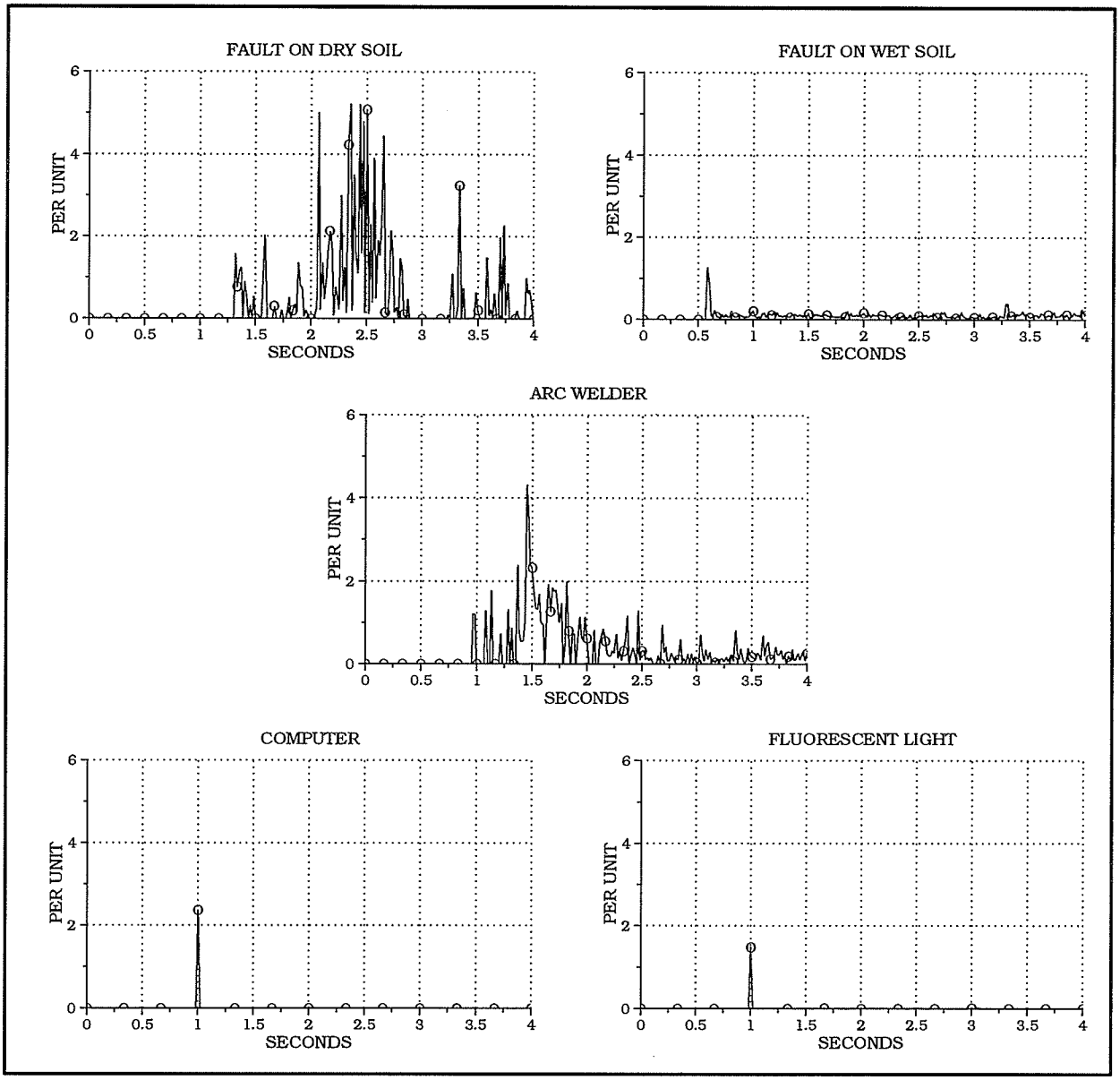

Figure 3.9. Even–to–odd harmonics power ratio.

3.5. Toward a Solution of the Problem

Patterns for current waveforms, for the various circumstances already described, are distinguishable by the human brain. Therefore, these patterns should be usable by an intelligent system as "fingerprints" for the corresponding disturbances in the system. The advances in neurocomputing

in pattern recognition motivated the use of artificial neural networks approach in designing an arcing fault detector, as described in the following chapters.

4 Artificial Neural Networks

Artificial neural networks (ANN) are computational structures modelled on the biological nervous system. The research on ANN is motivated by the fact that the brain outperforms modern digital computers in pattern recognition and classification of real world data in the presence of a noisy and distorting environment, and modelling the biological nervous system function using man–made machines increases understanding of that biological function. The following sections gives an overview on the background of artificial neural networks, details of the high impedance fault pattern identification ANN, and the back–propagation learning algorithm.

4.1. Overview of the Biological Nervous System *[27][28]*

4.1.1. Structure

The fundamental building block of the nervous system is the *neuron*: Fig. 4.1. The different shapes, sizes and lengths in which a neuron may exist are important to the function and utility of neurons. Neurons are imbedded in an aqueous solution of small ions. The selective permeability of a neuron to these ions establishes a negative electrical potential of some tens of millivolts. The *soma* is the round central cell body of the neuron (5 to 100 μm in diameter).

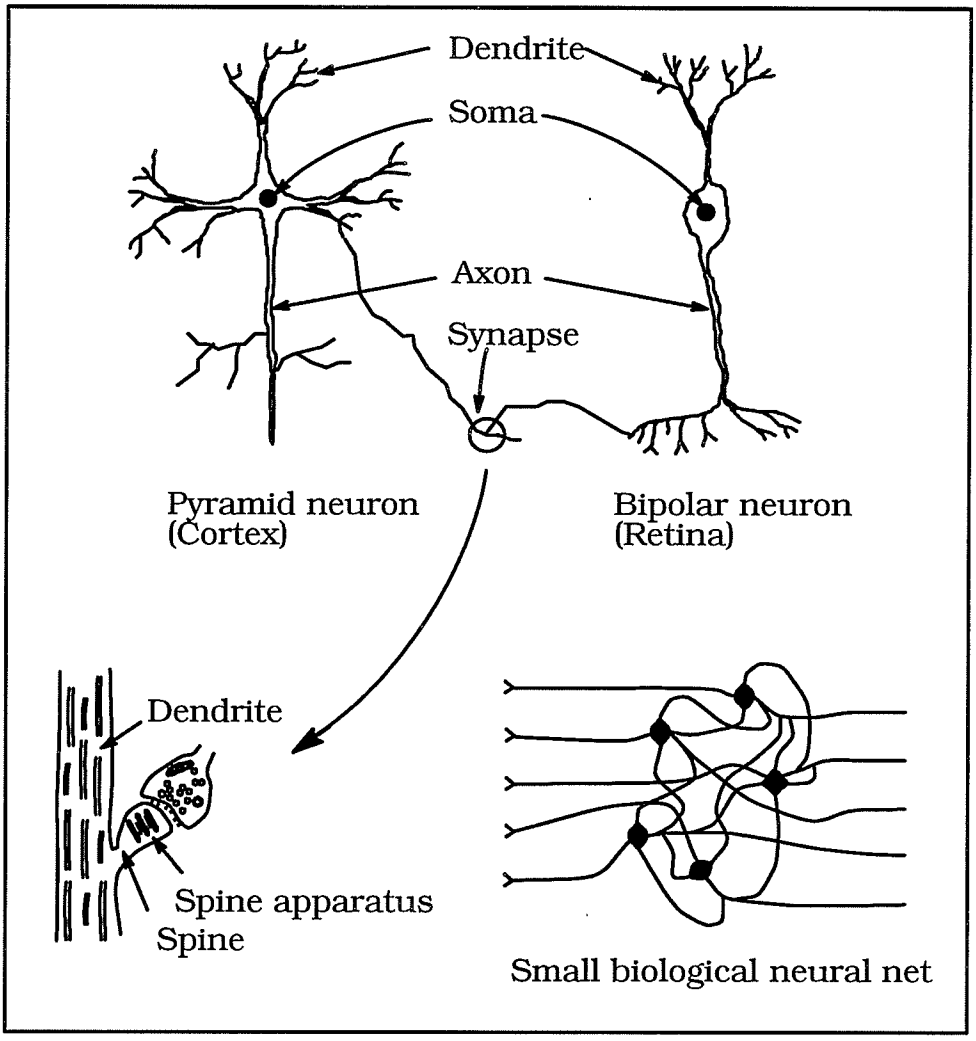

Figure 4.1. Fundamental components of biological neural networks. Top: two types of neurons. Bottom left: neuron–to–neuron connection. Bottom right: five neurons performing a small neural network [27].

The *axon* acts as the cell output. The axon, electrically active, is attached to the soma and produces the pulses emitted by the neuron. The electrically passive *dendrites* receive electrochemical input signals from other neuron axons. Axons and dendrites are of the order of 1 μm in diameter. The axon terminals form a connection, *synapse*, which almost touches the dendrites of other target neurons. The *synaptic gap*, shown in the bottom left of Fig. 4.1, is the space between the two cells: about 0.01 μm.

4.1.2. Communication

Neurons communicate to one another by transferring electrical energy along the axon to the axon terminals. *Neurotransmitters*, specific chemicals released by the pulse on the pre–synaptic side of the synapse, carries the signal across the synaptic gap to the next cell. On the receiving or the post–synaptic side of the synapse, the neurotransmitters bind to specific receptor molecules, opening ionic channels and changing the electrochemical potential. The magnitude of this change is determined by many factors local to the synapse such as the amount of transmitter released on the pre–synaptic side, and the number of receptor molecules on the post–synaptic side. These sites of neuron–to–neuron communication are capable of changing a dendrite's local potential in a positive or negative directions, depending on the pulse it transmits. These factors can change with time, thus, changing the effectiveness or "strength" of the synapse. This process takes about 0.1 to 0.2 ms of time, which is a long time by electronics standards. It is not known how many of the thousands of synapses on a cell are strong or even functional at a given time. Estimates of the number of active synapses necessary to cause a cell to "fire" range from a few to hundreds.

4.1.3. Processing

The synaptic potentials are combined in various ways. When the resulting potential is greater than a fixed threshold, the generation of an output signal is triggered by a special region near the cell body. The signal, called *action or axon potential*, is a large brief pulse, approximately 100 mV by 1 ms. The pulse propagates without attenuation down the axon and is delivered to synapses of the axon branches. Action potential pulses travel at velo-

cities up to 120 m/s, and can be produced at rates varying from 0 to 1500 times a second. Many action potential pulses are averaged over tens of milliseconds to determine the average firing rate on an axon. The resulting neural element computing time is of the order of tens of milliseconds. Nevertheless, the massive parallelism in the nervous system counters the relative slowness of individual neurons. Therefore, it is not surprising that humans can respond to complex stimuli in fractions of a second where the process requires hundreds of sequential steps.

This description of neural input/output function is reasonably accurate for a large number of neurons, but it has been revealed that much more complicated mechanisms exist. In addition, there are neurons without axons, synapses that are bi–directional, synapses onto other synapses and onto axons, and non–chemical electrical synapses.

The brain has a modular architecture. It performs its tasks by thousands of discrete structures of neurons. Each structure has its own particular type of neurons, pattern of connections, and role in brain function. For example, vision task is performed by systems composed of many interconnected structures, each serving a small specific subtask.

4.1.4. Statistics

The human cerebral cortex weighs three pounds, covers about 0.15 square meters, and is about two millimeters thick. It contains approximately 100 billion neurons. Each neuron has 1000 dendrites that form 100 000 billion synapses. This system functions at 10 000 billion interconnections per second. Brain capability is beyond anything which can be reconstructed or modeled. However, the possibility to understand how the

brain performs information processing could be modelled and ultimately implemented in hardware.

4.2. Artificial Neural Networks [27][28]

The origin of artificial neural networks, inspired by the biological nervous system, dates back to the late 1950s when a group of scientists took the approach of "how the brain did things" in an attempt to build an intelligent system. *An artificial neural network (ANN), or simply a neural net (NN),* is an information processing system (IPS) that extracts information from its input and produces an output corresponding to the extracted information. Neural nets are also called "connectionist models", "parallel distributed processing models", or "neuromorphic systems". Neural networks are distinguished from other fields with similar goals, e.g. artificial intelligence, by incorporating features of the biological nervous systems into its design. There are two classes of neural network models:–

1. ***Neurobiological models*** are computational models of biological nervous systems. Their object is to summarize and predict existing neurobiological and psychophysical data and behavior.
2. ***Computational models*** are biologically inspired models of computational devices with technology applications.

4.2.1. Analogy to Biological Nervous System

An artificial neural net is a system composed of many simple neuron–like processing elements called "artificial neurons", "nodes", or "units". The unit receives one or more inputs from other processing elements or from external

sources. These inputs are then modified by some weighting coefficients specific to each input according to a learning algorithm. The unit performs very little computation; typically, biasing the weighted sum of its input and passing the result through a nonlinearity. Figure 4.2 illustrates a single neuron ANN.

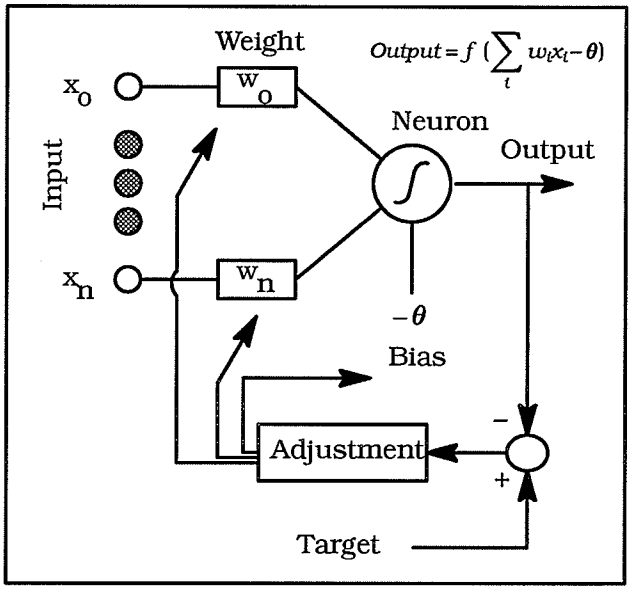

Figure 4.2. The learning mode of a single neuron artificial neural network. [27]

The benefits of ANN are high speed and fault tolerance due to massive parallelism, and adaptivity; i.e. neural nets can be trained, hence, their performance may improve with experience. Adaptation or learning is a high level function of an ANN. Vision tasks and speech recognition are low level functions a neural net can perform. The function of the neural net is determined by the connection topology and strengths.

4.2.2. Network Topology

Quite a few different neural network models have been developed to achieve human–like performance in the field of speech, vision and robotics.

Important models include the Hopfield network, single– and multi–layer perceptrons, the cerebellar model articulated controller (CMAC) network, the feature map network, Darwin III, and the silicon retina chip.

Connectivity patterns between nodes vary across ANN models. Nodes may be *locally–connected* to neighbors, *fully–connected* to all other nodes, or *sparsely–connected* to a few distant nodes. In addition, networks may be layered with exclusively *feedforward connections* from lower to higher layers as in multi–layer perceptron or provided with recurrent *feedback connections* as in fully–connected Hopfield network. The design of neural network feedback loops has implications for the nature of its *adaptivity/trainability.* The design of a network's interconnections has implications for its *parallelism.*

4.2.3. Training Algorithms

A neural network, as an information processing system, maps from the space of all possible inputs to the output space. The network is trained rather than programmed to perform the required processing. A *learning system* formulates the mapping function from the training examples presented to it. The *learning algorithm* is the functional specification of the transformation between inputs and outputs. *Implementation* of the IPS is the physical realization of the processing mechanism that runs the algorithm. One aspect of neural network research is to design new algorithms as machines that can solve problems that require "intelligent" analysis for their solution. Adaptive ANN can be trained using three types of training algorithms:–

1. ***Supervised training*** requires labelled training data and an external teacher who knows the desired correct re-

sponse and provides a feedback error signal. This is sometimes called *reinforcement learning or learning with a critic* when the teacher does not provide detailed error information. Multi–layer perceptron is an example of supervised trained networks.

2. ***Unsupervised training*** (*self–organization*) uses unlabelled training data and requires no external teacher. The feature map neural network, for example, forms internal clusters that compress the input data into classification categories.

3. ***Self–supervised training*** (*learning by doing or learning by experimentation*) is used by automata which monitor performance internally and require no external teacher. An error signal is generated by the system and fed back to itself. The correct response is produced after a number of iterations: Darwin III.

4.2.4. Artificial Neural Network Tasks

Neural networks can perform a variety of tasks. Figure 4.3 illustrates a number of these major tasks:–

1. ***Pattern Classification***

   Classifiers are trained with supervision using labelled training data to partition input patterns into a pre–specified number of groups or classes.

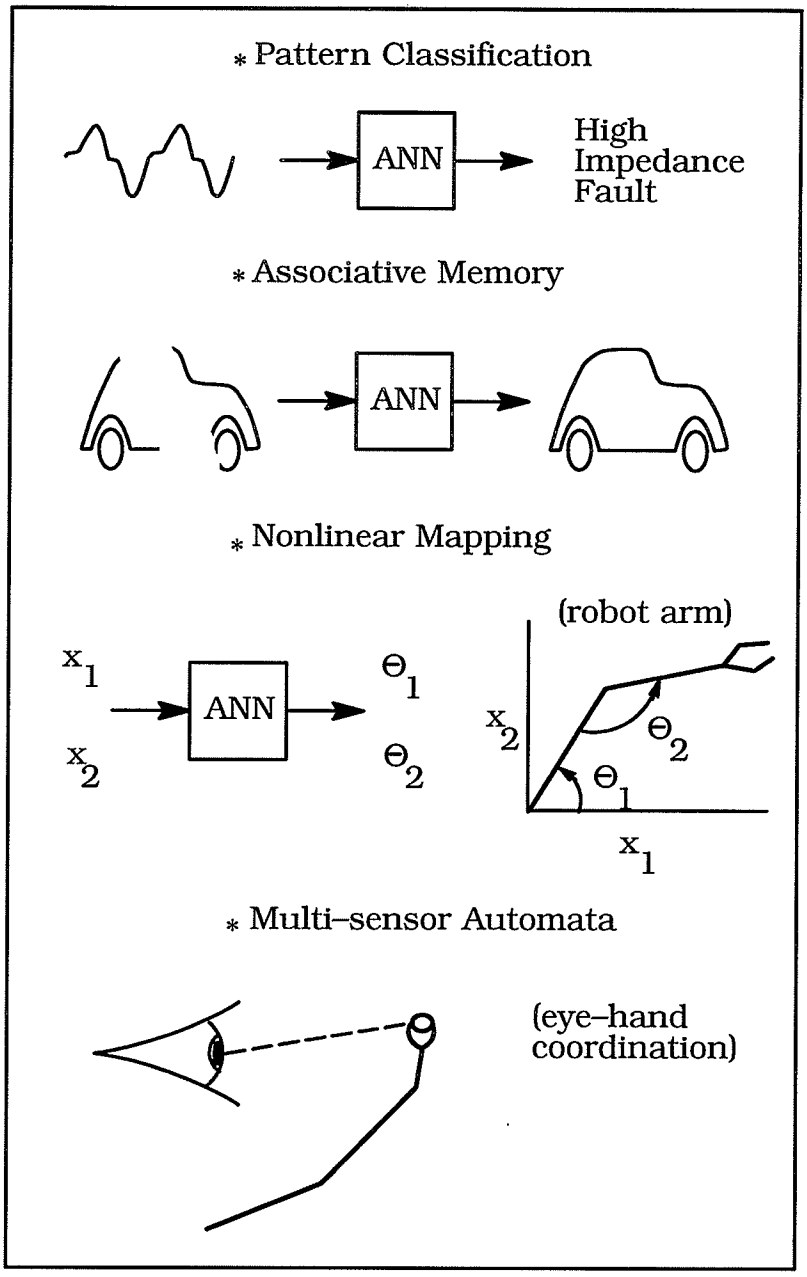

Figure 4.3. Illustration of a number of neural network tasks [27].

2. ***Associative Memory***

   A complete memory item is provided from a key consisting of a partial or corrupted vision of the memory.

3. ***Computation Problems***

   Nonlinear analog computation can be customized in ANN design

to solve constraint optimization problems.

4. ***Nonlinear Mapping***
   A vector of analog inputs can be mapped into an output vector using a nonlinear mapping function that can be learned from training data. This type of mapping is useful in robot control and nonlinear signal processing.

5. ***Multi–sensor Automata***
   Objects can be robot manipulated using visual and robot arm inputs in a number of complex, multi–module network automata.

4.3. Pattern Recognition Network

4.3.1. Training Patterns

The high impedance fault pattern recognition neural network was trained by a set of current patterns of arcing faults on dry and wet soil, computer, arc welder, fluorescent light, and sinusoidal loads. Figures 4.4 to 4.6 illustrate samples of the patterns used to train the network. A total of *twenty–six* patterns were used in training. *Sixteen* patterns of fault on dry soil and *two* patterns of fault on wet soil: Fig. 4.4. The similarity of the computer current to that of the HIF on dry soil suggested the use of the width of current conduction period as a feature that distinguishes a computer load from a fault current. The detection algorithm, described in chapter 5, performs a zero current search on the input current data in the preprocessing stage. Considering the period where the computer current waveform stays very close to zero, preprocessing could result in capturing the required zero crossing at different points on that waveform: Fig. 4.5. Therefore, four training patterns were used for the ANN to identify computer current at

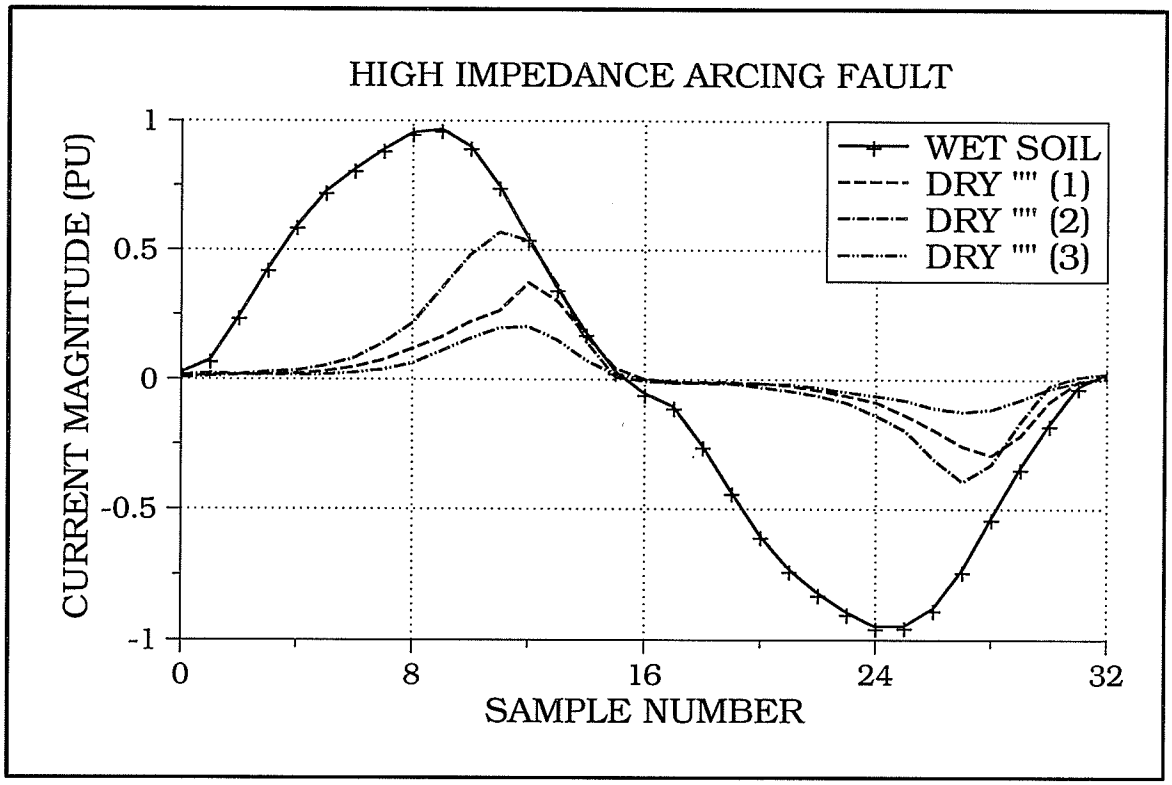

Figure 4.4. High impedance arcing fault training patterns.

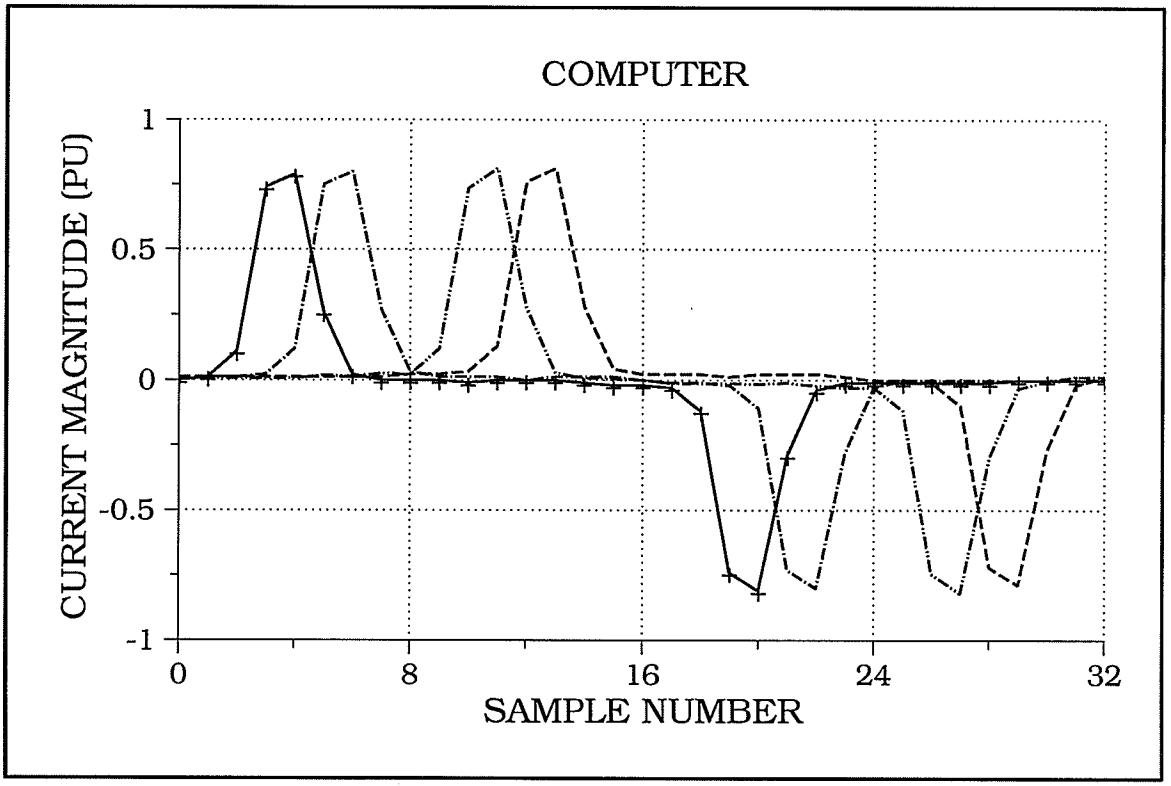

Figure 4.5. Computer current training patterns.

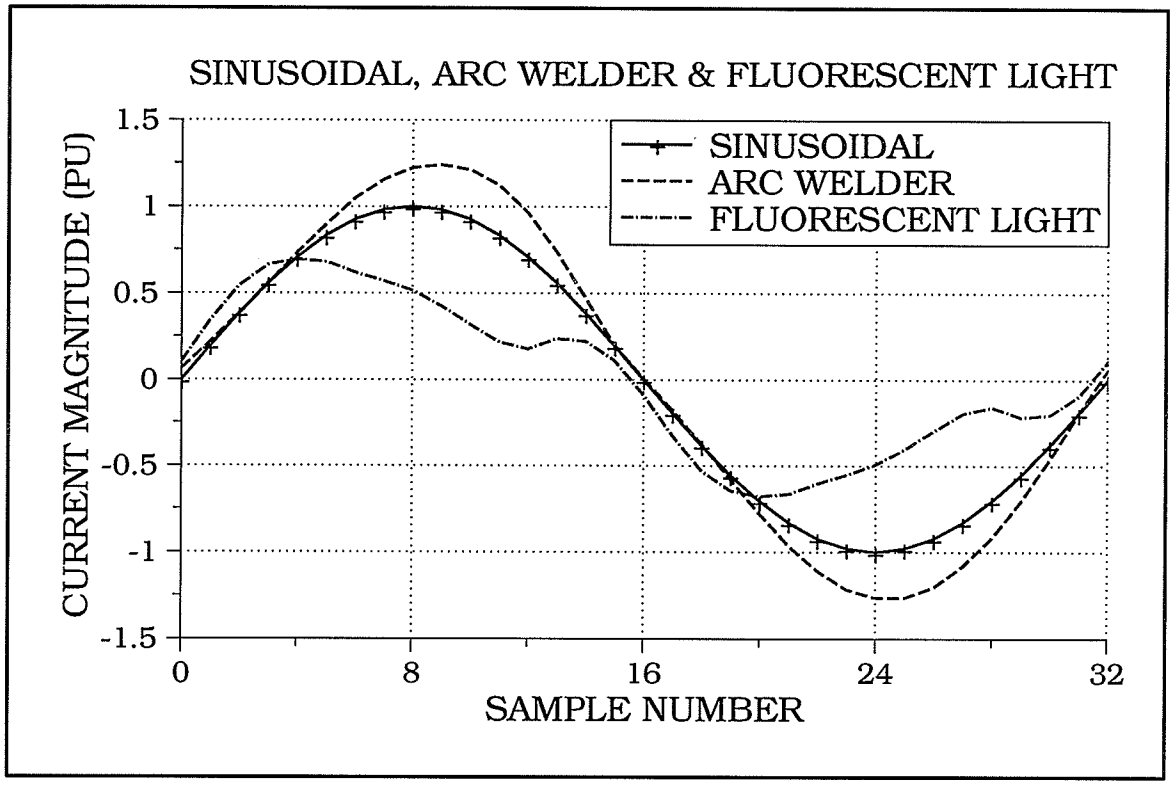

Figure 4.6. Sinusoidal, arc welder, and fluorescent light training patterns.

different phase displacement. The network was trained by *two* patterns of the arc welding machine current, *one* pattern of fluorescent lighting load, and a sinusoidal current pattern to reinforce the network experience in distinguishing faults on wet soil from normal loads: Fig. 4.6.

4.3.2. Network Architecture [27][28][29][30][31]

The Multi-layer Perceptron

A multi-layer perceptron (or back-propagation network)—a fully-connected neural network structured with at least three layers of nodes (input, hidden, and output), and with only feedforward connections between the adjacent layers—has been used to classify arcing fault currents as distinct from fault-like and normal load currents. In feedforward nets, all input is received in one layer, and the resulting signals propagate forward, one layer

at a time, until the signal reaches the last layer. The hidden layer(s) of the network extract the *features* found in the input. The *features* are the correlation of activities among different input nodes. Therefore, an abstract higher–level representation of the input information is presented in the hidden layer(s). The output layer responds to the presence of features in the pattern rather than the pattern itself. As the network becomes able to respond to the features of different input patterns, it develops the ability to *generalize*. The complex decisions capability an ANN acquires, therefore, is due to the feature detection and generalization abilities which are trained into the hidden layer(s) nodes.

The Meso–Structure

The architecture of the pattern recognition network used in arcing faults detection is shown in Fig. 4.7. The input vector is composed of 33 elements. The first 32 elements are analog inputs, of values between "0" and "1", that represent the instantaneous values of the sampled line current per cycle, starting at the current zero crossing of the positive half–cycle. The last element in the input vector holds a binary value. This element carries a value of "1" if the width of the current conduction period, defined as the number of current samples of magnitude > 30% of the peak instantaneous current per–cycle, is less than eight samples per–cycle, otherwise "0".

The hidden layer of the network is composed of six hidden neurons. This layer undertakes the nonlinear mapping between the input and the output. Selection of the optimal number of hidden neurons to provide optimum network performance, taking into account the training patterns and network

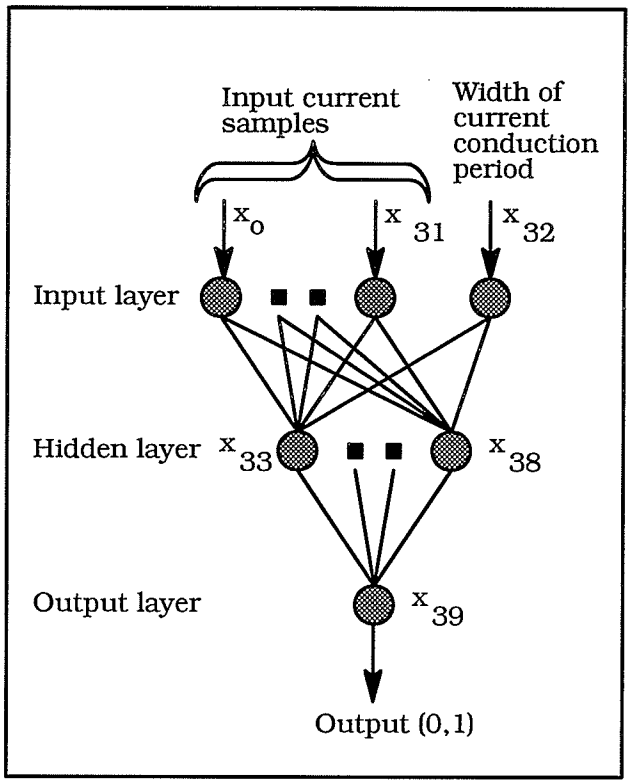

Figure 4.7. Architecture of the pattern recognition network.

topology, is still an open issue. Nevertheless, the guidelines for determining the number of hidden neurons in a binary network was adopted as a starting point for this hybrid input network. For an arbitrary training set with *N* training patterns, a multi–layer neural network with one hidden layer and with *N–1* hidden layer neurons can exactly implement the training set. A minimum number of hidden units should be selected such that they respond to the features in the input patterns, and reduce the computational time needed for training without being too small to not converge. According to these criteria, the number of hidden layer units should be either "4" or "5" depending on whether the dry and wet arcing faults are considered as one or two different patterns. The process of selecting the number of hidden neurons started by training then testing a network of "4" hidden units. The

results showed that the detection algorithm is confused between the arc welder load and faults on wet soil. The use of two or three hidden nodes yielded no advantage. The use of "5" hidden nodes showed a much better performance than the "4" hidden nodes ANN. Further analysis using different number of hidden neurons suggested that the best performance is achieved with "6" hidden nodes. The results will be demonstrated in the next chapter.

The output layer has one neuron. The target output of the network in the HIF event is "1", otherwise "0". Since the neuron output function is a sigmoid, the "network score" would be between these two values.

Neuron Transfer function

The back–propagation learning rule, derived in the following section, requires that the derivative of the *neuron transfer function* exists. Furthermore, this *activation function* should provide a threshold that if the unit activation exceeds a threshold by a sufficient margin it will always attain a value of "1". If it is far below threshold, it takes the value "0". Therefore the function should be continuous, nonlinear, and asymptotic for both infinitely large positive and negative values of *activation* (the sum of neuron inputs). The *logistic* or *sigmoidal* transfer function, equation (4.5), satisfies these conditions as shown in Fig. 4.8.

$$o = \frac{1}{1 + e^{(-a/T)}} \tag{4.5}$$

where o is the neuron output, a is the neuron activation, and T is the sigmoid temperature or degree of nonlinearity.

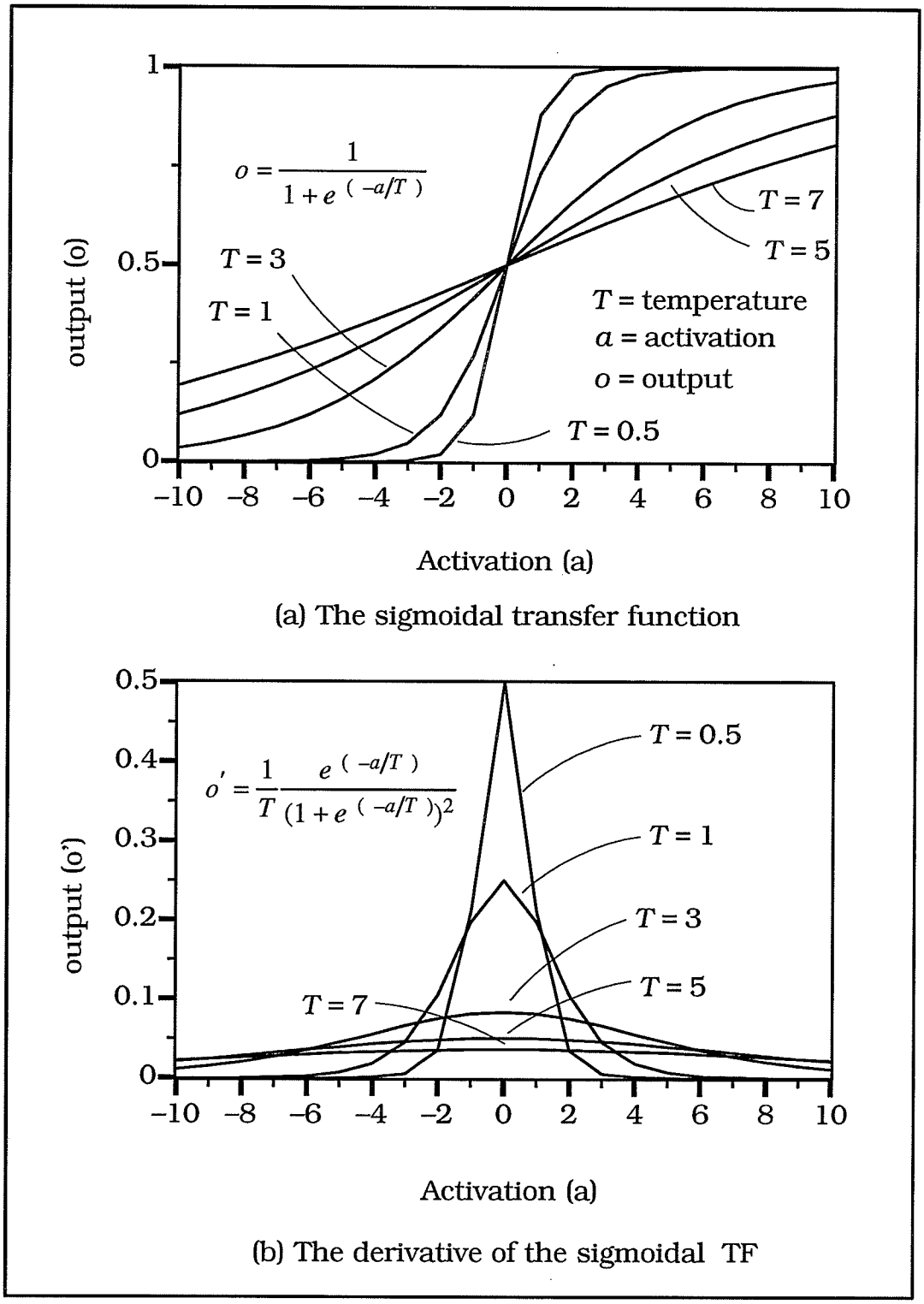

Figure 4.8. The neuron sigmoidal transfer function and its derivative.

4.3.3. Learning Rule *[28][31][32][33][34]*

The learning rule (algorithm) of the network is the set of equations that modifies some of the weights and biases of processing elements in response to input signals and values supplied by the transfer function. Weight adjustment is required to reduce the difference between the actual output and the desired output. The response of the processing elements to input signals, therefore, changes over time. The network converges at a set of connection weights which minimizes the error for recognizing all patterns in the training set. These weights are not unique; there will exist multiple sets, of similar weight values and infinite range of connection weights and biases, with workable answers. There might not be any single best answer.

The multi–layer perceptron, trained with the back–propagation algorithm, undergoes supervised training. Networks undergoing supervised learning have high performance to recognize patterns similar to those it has learned. The training sets consist of a number of training vector pairs. Each training vector pair is composed of the input pattern data (33 elements in our case), and the desired target for this pattern ("1" for fault, and "0" for no–fault).

The Generalized Delta Rule

The back–propagation algorithm, known as the *Generalized Delta Rule*, is an iterative gradient search algorithm (similar to Newtons method for finding zero–crossing of curves) that minimizes the cost function equal to the mean square error between the desired and the actual output of a multi–layer feedforward perceptron. The total error E is given by

$$E = \sum_p E_p = \frac{1}{2}\sum_p \sum_j (t_{pj} - o_{pj})^2 \quad (4.6)$$

where E_p is the error for pattern p. The index p ranges over the set of training patterns and j ranges over the set of output units. The target t_{pj} is the desired output for the j^{th} output node when the p^{th} pattern has been presented. The actual output of the j^{th} node when the p^{th} pattern has been presented is o_{pj}. Since the neuron transfer function is a sigmoid, the neuron output o_{pj} is

$$o_{pj} = \frac{1}{1 + e^{(-a_{pj}/T)}} \quad (4.7)$$

where the selected sigmoid temperature T is "1" and a_{pj} is the activation of node j for a given pattern p,

$$a_{pj} = \sum_i w_{ij}\ o_{pi} + \theta_{pj} \quad (4.8)$$

where w_{ij} are the weights connecting nodes i to node j, θ_{pj} is node j^{th} bias, and $o_i = i_i$ if node i is an input unit (propagation from input layer nodes is linear). For the network shown in Fig. 4.7 and the training set used, the indexes are $i = 0,1,..,38$, $j = 33,34,..,39$ and $p = 0,1,..,25$.

Because there are many different patterns to which the error function should be minimized, it is not simple to set the derivative of E with respect to w_{ij} to zero, and solve for $w_{ij\,(min)}$. The search of connection weights that minimizes the cost function E can be demonstrated using Fig. 4.9. This figure simplifies a high–dimensional error space by one dimensional slice. The tangent to the error curve can be used to change w_{ij} so that it approaches

the valley of the cost function. The system will follow the contour of the error surface—always moving downhill in the direction of steepest descent. The resulting *global* minimum (E_{min}), where the total error reaches a minimum for all patterns, depends on the random starting state of the training weights and biases. Therefore, it is possible that the system may have different *global* minima. Some of these minima could be deeper than others; the *best* possible solution to the problem may not be at hand. In some instances a *local* minimum may occur (E'_{min}); the connection weights do not minimize the error of all training patterns, e.g. using "2" or "3" hidden nodes in the HIF pattern recognition network. This problem is common in networks with few hidden neurons, and rare in networks with many hidden neurons.

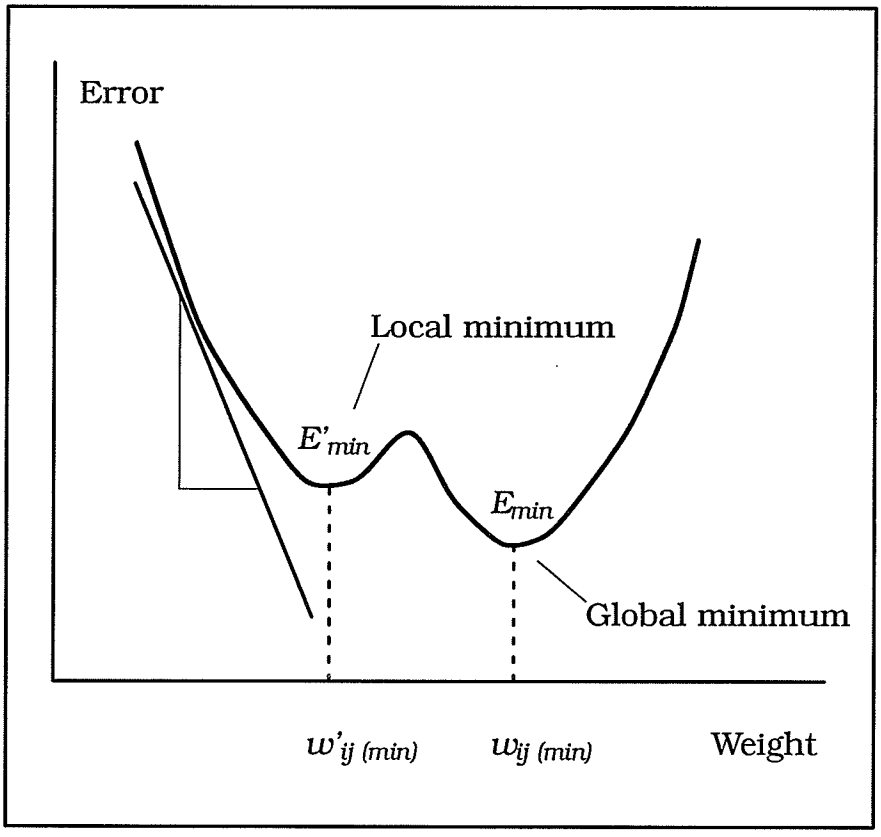

Figure 4.9. One dimensional slice of the error surface. Search for minimum sum of squared errors [28].

The training rule allocates an error term to each connection weight. The error is expressed in terms of a function Delta (Δ) applied to weight w_{ij} after each presentation of a training pattern. This function is proportional to the negative of the derivative of the error with respect to the connection weight,

$$\Delta_p \; w_{ij} \; = \; -\eta \frac{\partial E_p}{\partial w_{ij}} \tag{4.9}$$

The constant of proportionality η is the learning rate, a constant between zero and one, that affects the step change in weights. This rate is practically chosen to be as large as possible to offer the most rapid learning without leading to oscillation. To avoid the occurrence of local minima in some problems, very small values of η could be used (e.g. 0.1).

To calculate the error term, the chain rule from differential calculus is applied to the partial derivative on the right–hand–side (RHS) of equation (4.9) ,

$$\frac{\partial E_p}{\partial w_{ij}} \; = \; \frac{\partial E_p}{\partial a_{pj}} \frac{\partial a_{pj}}{\partial w_{ij}} \tag{4.10}$$

From equation (4.8),

$$\frac{\partial a_{pj}}{\partial w_{ij}} \; = \; o_{pi} \tag{4.11}$$

Define,

$$\delta_{pj} \; = \; -\frac{\partial E_p}{\partial a_{pj}} \tag{4.12}$$

Apply the chain rule to the RHS of equation (4.12),

$$\delta_{pj} \; = \; -\frac{\partial E_p}{\partial a_{pj}} \; = \; -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial a_{pj}} \tag{4.13}$$

From equation (4.7),

$$\frac{\partial o_{pj}}{\partial a_{pj}} = \frac{1}{T} o_{pj}(1 - o_{pj}) \tag{4.14}$$

For the node(s) in the output layer ($j = 39$), using E_p definition in equation (4.6),

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) \tag{4.15}$$

Hence, for the output node(s), the difference between the *actual* value of the output node(s) and the *desired* value of the output node(s) is used to drive the change in weights at the output layer,

$$\delta_{pj} = \frac{1}{T} o_{pj}(1 - o_{pj})(t_{pj} - o_{pj}) \tag{4.16}$$

The adjustment in the hidden layer(s) differs from that at the output layer because the target output of the hidden nodes is not known. The values of δ at the output node(s) and the hidden–to–output weights are used to help determine the changes made to the input–to–hidden weights.

For the hidden layer nodes ($j = 33,34,..,38$), use the chain rule again to write

$$\frac{\partial E_p}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial a_{pk}} \frac{\partial a_{pk}}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial a_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ik} o_{pi} =$$

$$= \sum_k \frac{\partial E_p}{\partial a_{pk}} w_{jk} = \sum_k \delta_{pk} w_{jk} \tag{4.17}$$

where $i = 33,34,..,38$, and $k = 39$ (one output node).

For the hidden nodes, therefore,

$$\delta_{pj} = \frac{1}{T} o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{jk} \tag{4.18}$$

The mathematical expression of the Generalized Delta Rule to change connection weights for a given pattern p is

$$\Delta_p \; w_{ij} \; = \; \eta \delta_{pj} o_{pi} \tag{4.19}$$

where δ_{pj} is expressed for the output layer nodes by equation (4.16), and for the hidden neurons by equation (4.18). There is no attributable error to the input nodes. Figure 4.10 illustrates a block diagram clarifying the Generalized Delta Rule and the back–propagation of error signal during training. Weights are adjusted either after each pattern has been processed, or after an entire processing epoch where the total error derivatives are accumulated. In the latter mode the new weights and biases are computed by

$$w_{ij}(new) \; = \; w_{ij}(old) + \sum_p \Delta_p \; w_{ij}(old) \tag{4.20}$$

and

$$\theta_j(new) \; = \; \theta_j(old) + \sum_p \Delta_p \; \theta_j(old) \tag{4.21}$$

for $i = 0,1,..,38$, $j = 33,34,..,39$ and $p = 0,1,..,25$.

Impact of Neuron Activation Function on Learning

The neuron transfer function helps the learning law work effectively. The derivative of the function is always positive, having its peak value at "0" activation and is close to "0" for large positive and negative inputs: Fig. 4.8 bottom. Since weight change is proportional to activation function derivative, the weight change is large for inputs in the mid range (near "0"). The learning rule is trying to bring the neuron to one of the stable states, "0" or "1", where the derivative of the activation and hence the error are close to "0". However, some difficulties could arise in learning if a weight change is slower than desired.

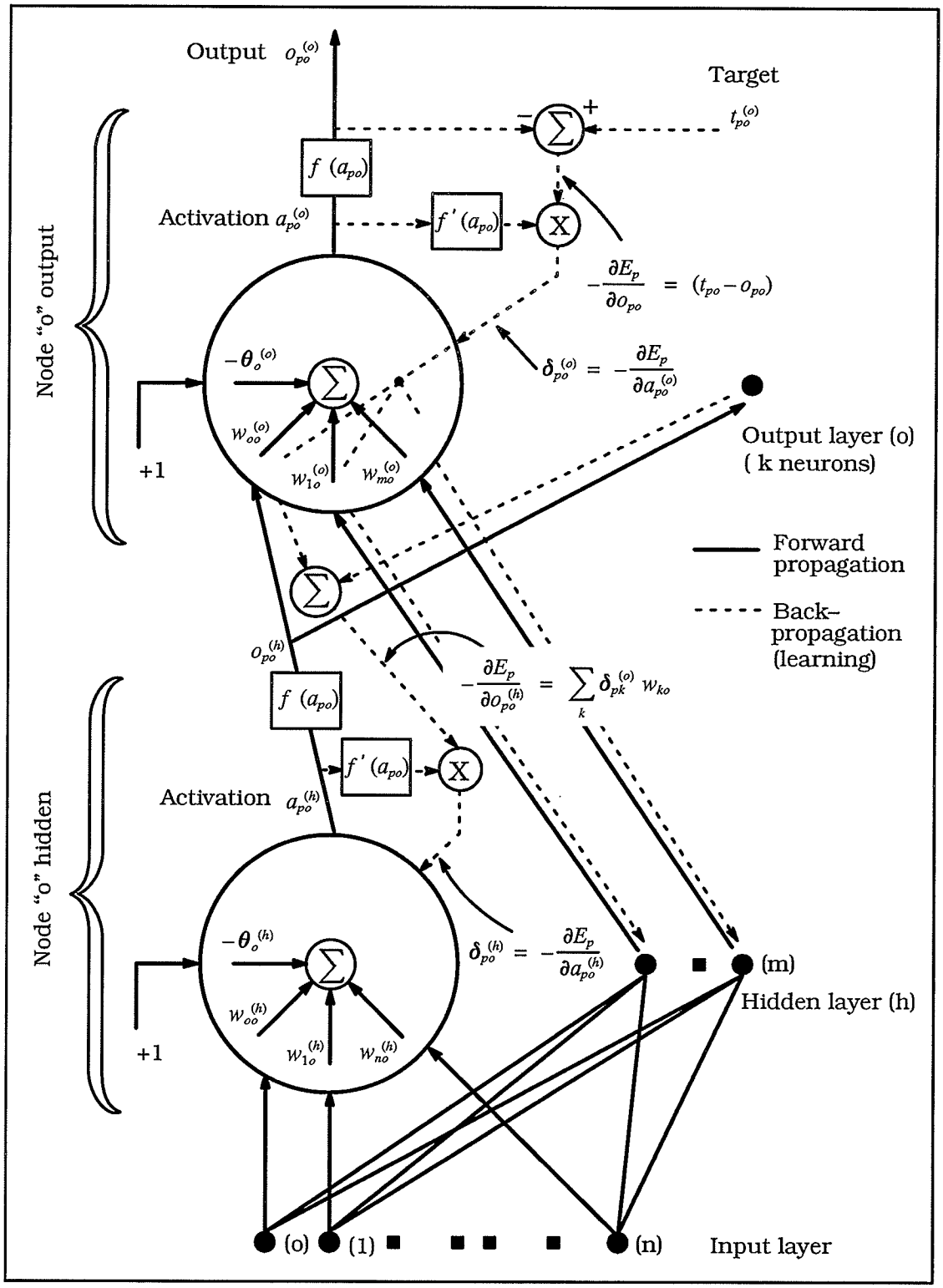

Figure 4.10. Block diagram illustrating error allocation in the back-propagation network [34].

Training Procedure

The software of the back–propagation program [33] has been used to obtain the weights and biases of the pattern recognition network. Small random weights and biases are initially selected as illustrated from the screen display shown in Fig. 4.11. A factor of "100" is included in the weights and the nodes output at each layer. The network is trained by presenting all training data repeatedly. Weight adjustment is done after processing each pattern. Once the connection weights between the hidden and the output layers, along with the bias of the output node, were adjusted, the connection weights between the input and the hidden layers, along with the biases in the hidden layer, are adjusted.

Learning by pattern was the selected mode of training. This training mode showed better results and performance over learning by epoch. It was concluded that the pattern learning mode is more suitable for learning current waveform shapes than the epoch learning mode.

The suitable learning rate was selected by running the training algorithm at learning rates of 0.1, 0.25, 0.5, and 0.75. Training with $\eta = 0.75$ produced oscillation during the learning process: the error function was maximized rather than minimized; the weights remained unchanged; and the system did not learn. The same observation was valid when the network was epoch trained at a learning rate of 0.5. Reducing the learning rate escaped this problem: the network reached a global minimum, and the network performance was almost identical in all cases. The training time required to reach a total error of 0.001 was found to be 3188, 1024 and 520 for $\eta = 0.1$, 0.25 and 0.5 respectively—the average learning time for a back–propagation

```
disp/ exam/ get/ save/ set/ clear cycle do log newstart ptrain quit
reset run strain tall test

epoch     0   tss  0.0000   pattern   0  gcor     0.0000
cpname        pss  0.0000

INPUT :

      X0    X1    X2    X3    X4    X5    X6    X7 ———X32
       0     0     0     0     0     0     0     0     0
 0     0     0     0     0     0     0     0     0     0
 0     0     0     0     0     0     0     0     0     0
 0     0     0     0

HIDDEN :
                X33————————————————————————————X38
                 0     0     0     0     0     0

OUTPUT :
                           X39
                            0
TARGET :
                            0

WEIGHTS :

Input — Hidden (not all)

-40    36    48    26   -34   -44     2    16     8    22
 23   -22   -21   -46   -13   -40    40   -47    -7    23
 10   -14    20   -26    24    14   -15    44    48   -29
 14    -9   -19    23    -3     2    34     3     7    33
-26     8    16     0    27    44     9    30    -8   -43
-45    29     6    42     6   -44    31    11   -37    42
 32     4    10    17    10   -28   -24   -28   -26    21
  4   -26   -43    -8    -6    27    39   -48    25    -9

Hidden — Output

44     1     9    39     9    -9
```

Figure 4.11. Startup of network training.

network is between 100 to 10 000 times. As a result of this comparison $\eta = 0.5$ was selected as the learning rate of the network to calculate the weights and biases.

Network training with different initial weights showed that the network comes to optimum settings, within 400 to 700 processing steps, when the total sum of squared errors is in the range between 0.002 to 0.001, at the specified η . Figure 4.12 shows some of the weights of the trained network. It is noticed that the hidden–to–output weights of nodes 33, 34, 36 and 38 dominate the others in the same connection. The network is now trained and ready for operation. The connection weights are fixed. Back–propagation refers only to the learning stage. No information is passed backward during network operation.

Input Pattern Features

The activity of the hidden layer nodes is examined in Figures 4.12 to 4.17 to determine hidden nodes response to different input patterns. For high impedance faults, nodes 33, 36 and 38 are highly active; particularly with fault on dry soil. The same nodes are also active, to a much lesser extent, to arc welder current pattern: Fig. 4.14. In each case, the output of these nodes is almost equal to one another.

Nodes 34 and 36 are the active hidden units when the input pattern is the computer current: Fig. 4.15. The weight connecting node 34 to the output node is highly negative, Fig. 4.12, inhibiting the total output of the network. The activation of node 34 and de–activation of nodes 33 and 38 could be due to the logic incorporated in the last element in the input vector which

```
disp/ exam/ get/ save/ set/ clear cycle do log newstart ptrain quit
reset run strain tall test

epoch  520   tss  0.0000   pattern   0  gcor     0.0000
cpname d_0   pss  0.0000

INPUT :

      X0   X1   X2   X3   X4   X5   X6   X7 ———X32
      51   52   52   52   52   54   56   60   65
71    79   85  100   90   72   55   50   48   48
48    48   46   44   41   38   31   24   15   11
21    38   49    0

HIDDEN :
                X33————————————————————————————X38
                 97     0     5    99     8    98

OUTPUT :
                          X39
                           99
TARGET :
                          100

WEIGHTS :

Input — Hidden (not all)

 -5  -205  -302  -266  -190  -167  -105    65   254   359
 48   182   203   272   239   159   111   -67  -251  -312
 -2   -38     1   -51     0    -9    -9    27   -23   -38
-40  -202  -263  -219  -211  -222  -165   -69   197   280
-24   -74   -41   -23   -65     1   -40    -1    39    19
  4  -259  -235  -201  -217  -176  -161    30   158   275
 93  -107  -106  -215  -270  -205   -60   158   253   322
275   246   271   135     4   -70   193   310   367   478

Hidden — Output

 413  -673    59   520    93   484
```

Figure 4.12. End of network training and result for a fault on dry soil.

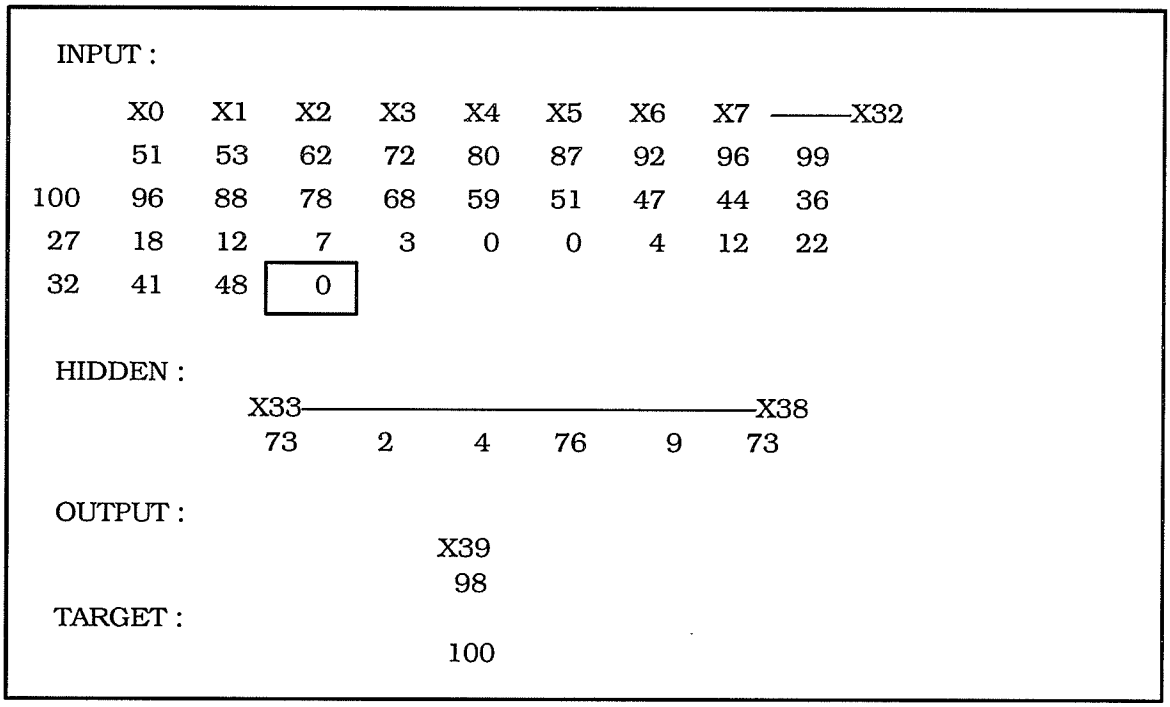

Figure 4.13. Network response to a pattern of fault on wet soil.

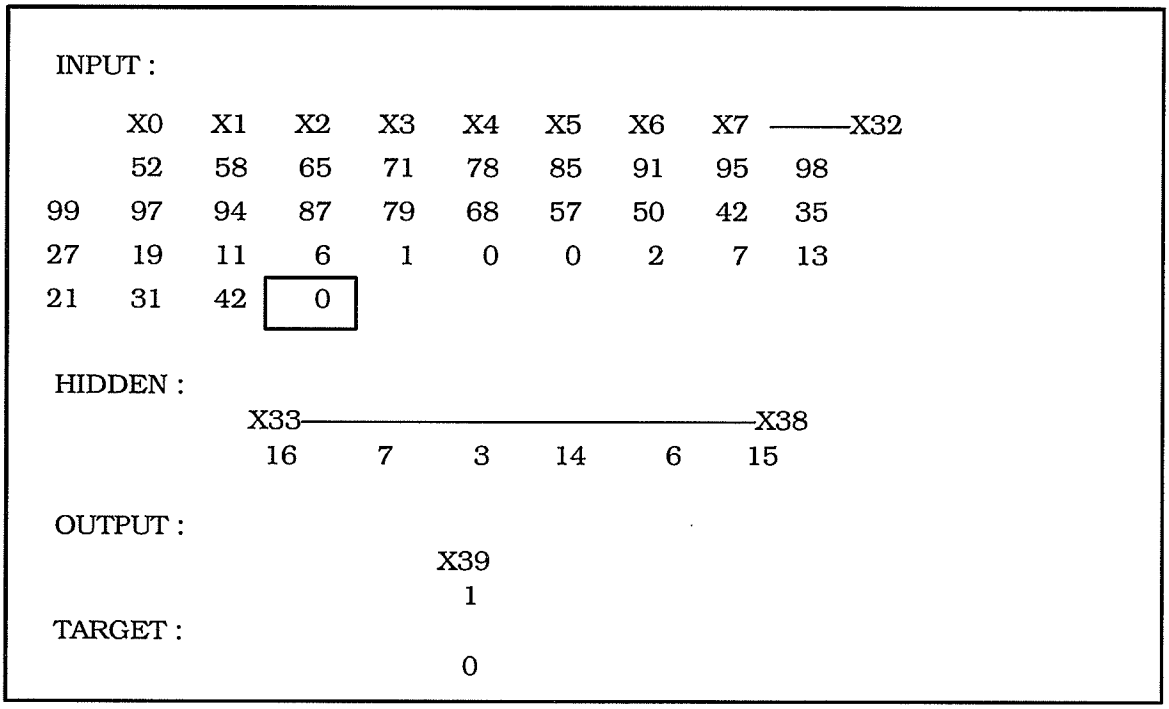

Figure 4.14. Network response to a pattern of arc welder current.

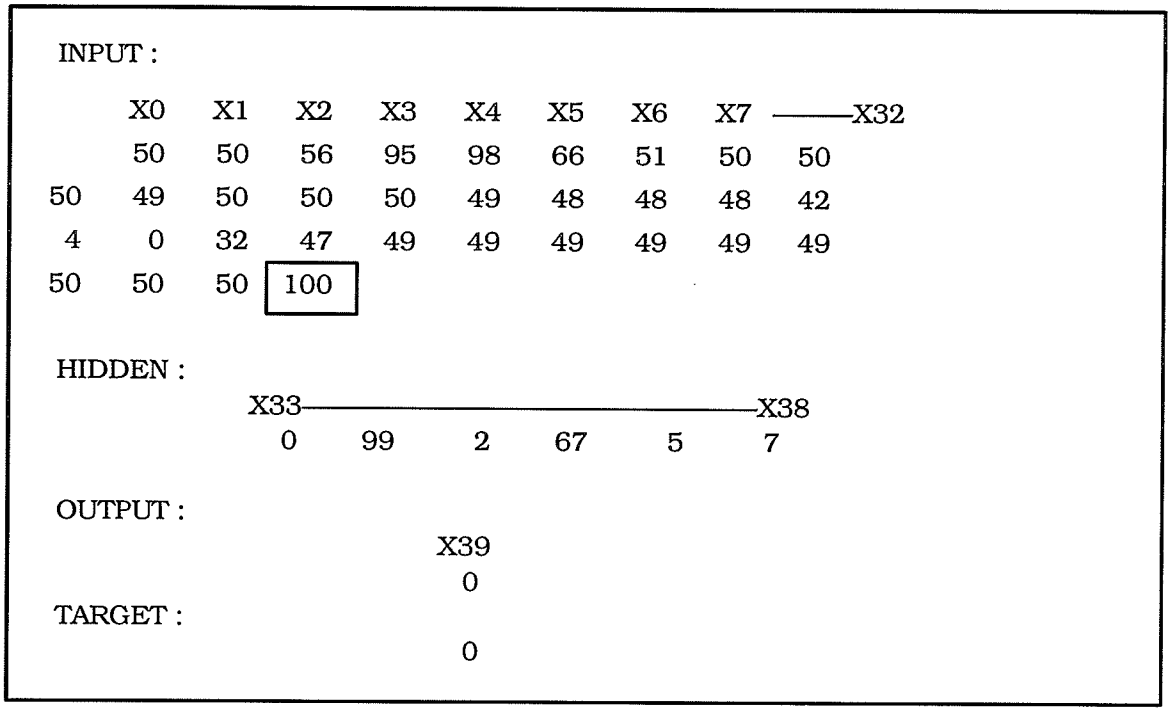

Figure 4.15. Network response to a pattern of computer current.

accounts for the width of the current conduction period per–cycle. Without this feature having been presented, the network might have responded to the computer current as to the HIF.

The activity of node 34 and the associated hidden–to–output weight appears to be important in separating fault–like and normal loads from fault current. This can be observed for the fluorescent light, Fig. 4.16, and the sinusoidal load: Fig. 4.17. For the latter load, the activity of node 34 decreased. This could be due to the close similarity to the fault on wet soil waveform.

It is concluded that nodes 33, 36 and 38 describe features local to current waveforms having off–current flow periods, as HIF, arcing loads, and some electronic devices. The other hidden nodes may carry other aspects

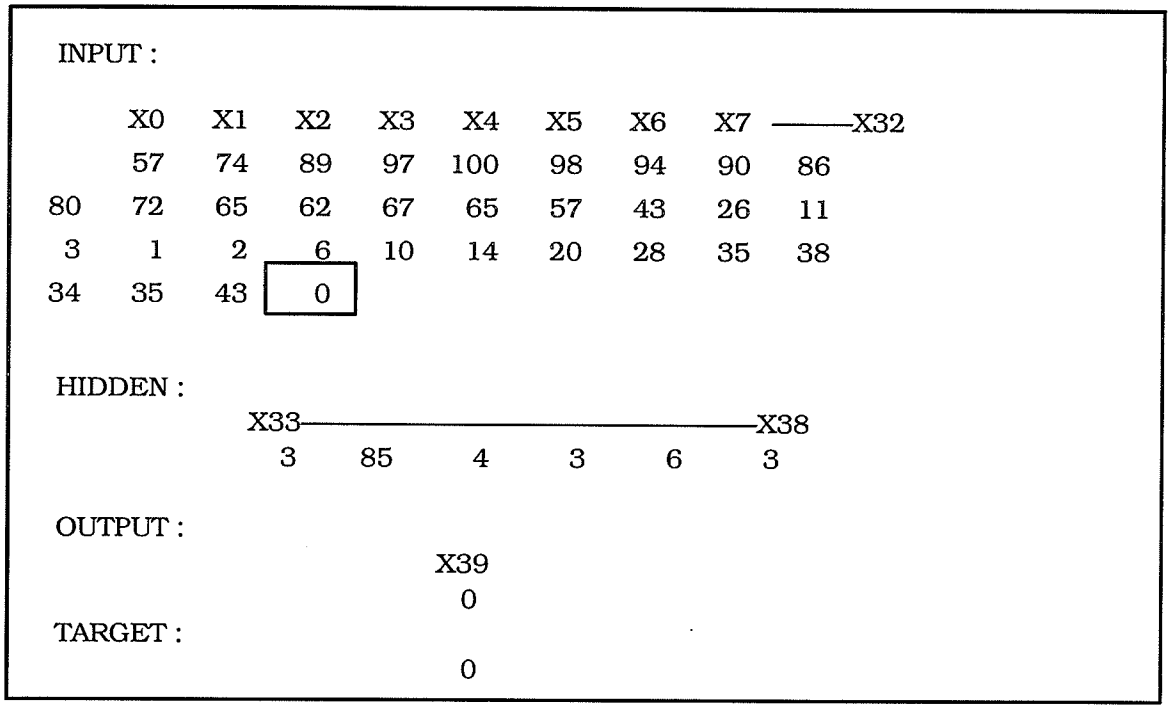

Figure 4.16. Network response to a pattern of fluorescent light load.

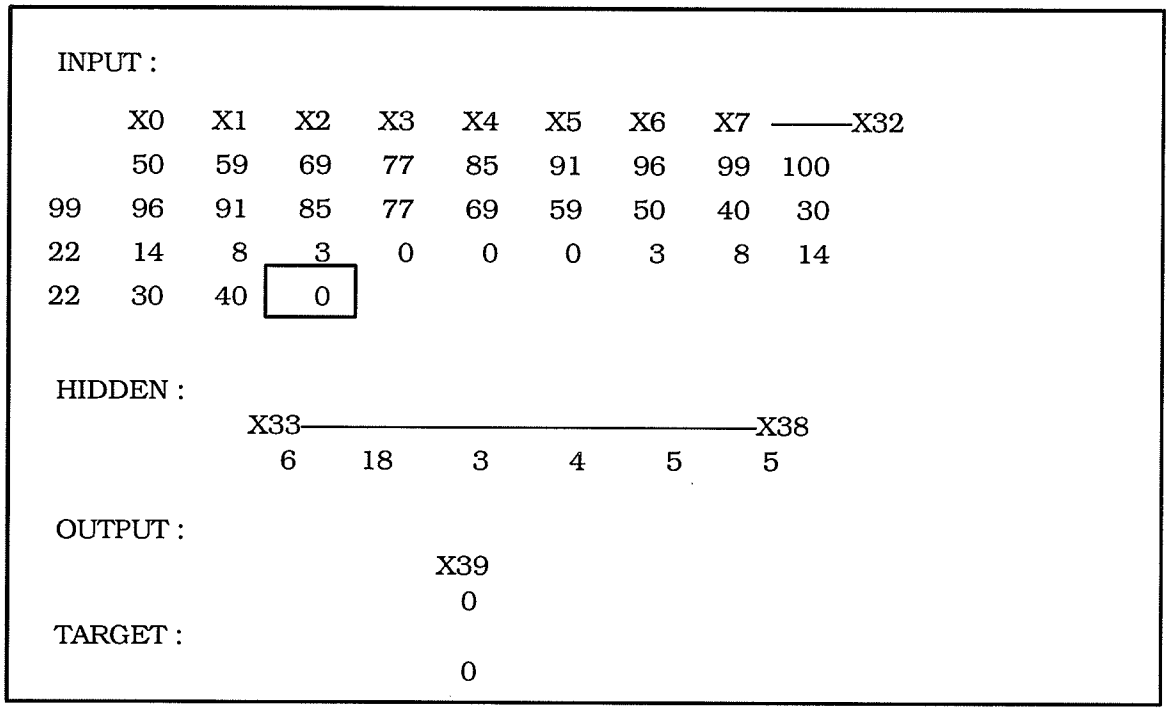

Figure 4.17. Network response to a pattern of sinusoidal load.

of the overall patterns. The exact features presented in the hidden layer are difficult to figure out.

5 Arcing Faults Detector

The review on high impedance faults detection, chapter 1, showed that no single detection method can detect all electrical conditions resulting from downed conductor faults. Furthermore, there are loads that imitate high impedance faults. Some detection methods might not distinguish between faults and fault–like loads: chapter 3. Quite a few of the detection methods require extensive computations in the preprocessing stage to extract the features of the input signal(s). A criterion is then applied to obtain the detection parameters.

The design of a *reliable* high impedance fault detector would include a number of detection methods to provide the required *dependability,* as well as the proper means to ensure its *security.* This chapter proposes a novel algorithm to detect high impedance arcing faults [35], motivated by the advances in neurocomputing in pattern recognition, that uses a simple preprocessing algorithm.

5.1. Relay Platform [36]

The proposed arcing fault relay will be a part of an integrated relaying scheme that carries out a number of protective and line monitoring tasks. The general layout of this integrated system is illustrated in Fig. 5.1. Input analog signals are conditioned and filtered. The signals are sampled at a

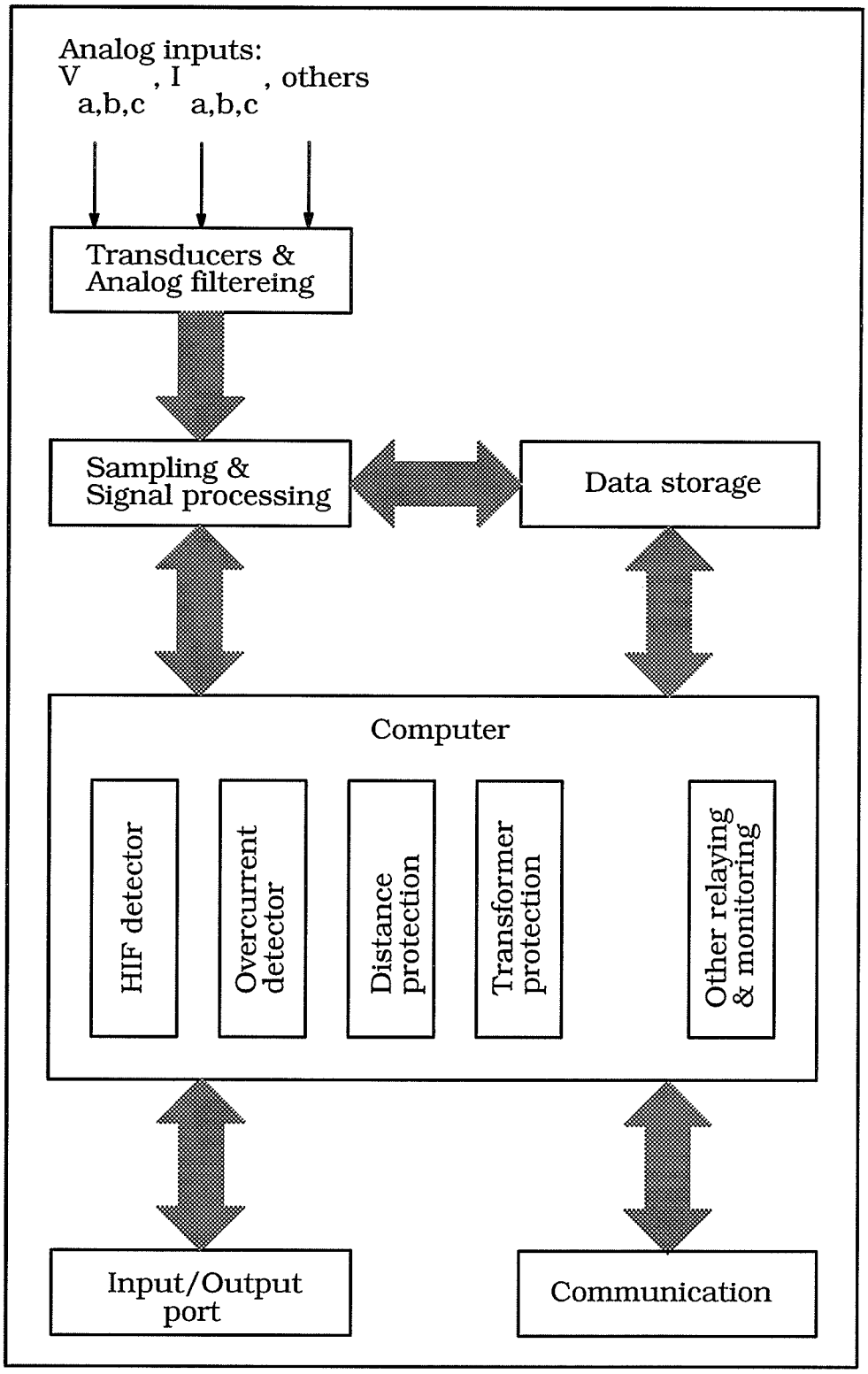

Figure 5.1. Integrated relaying and monitoring system [36].

preselected sampling rate. A digital signal processing (DSP) chip performs calculations on the sampled signals to obtain information such as the root–mean–square (rms) value, sequence components, and harmonics. The relay designer may use these data in full or in part in his/her relay algorithm. The different relaying functions are implemented as subroutines in a software package. The main software resides in a computer. At the occurrence of a system disturbance the system performs the required relaying to identify the type of disturbance. Digital signals are sent through the input/output (I/O) port for tripping or blocking functions. Information could be transmitted via communication links to the dispatch and control center. The arcing fault detector is one algorithm among others in this system. The following sections reveal the details of design and testing of this algorithm.

5.2. Detection Algorithm

The flow diagram of the high impedance arcing fault detection algorithm is shown in Fig. 5.2. The analog line current is low–pass filtered and sampled at the rate of 32 samples–per–cycle. The algorithm processes the data as follows :–

Startup

One cycle of the normal load current is stored in a buffer as the reference load current pattern. Its *rms* value is calculated: $I_{rms_{ref}}$.

Disturbance Trigger

The algorithm compares the *rms* value of the following cycles to $I_{rms_{ref}}$. When the new value is sufficiently different from the reference value, the

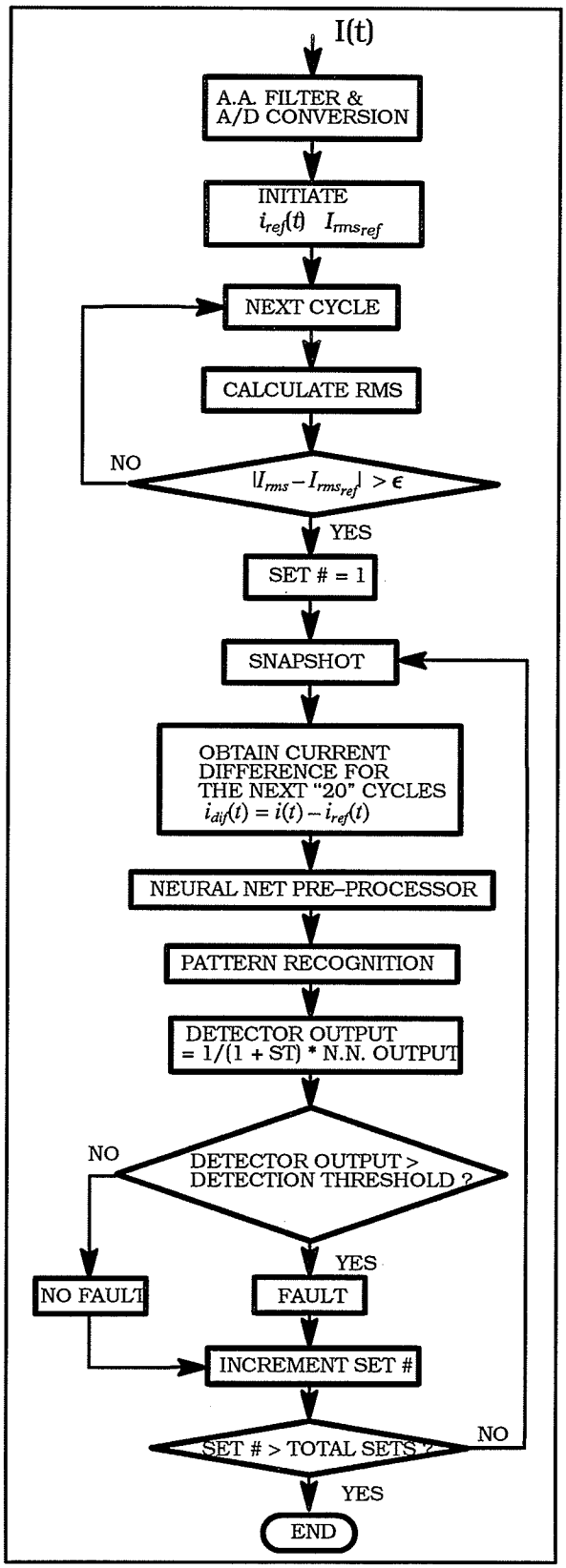

Figure 5.2. Flow diagram of the arcing fault detector.

detector starts to take twenty–cycle snapshots of the input current. Current fluctuation within ±20 % of $I_{rms_{ref}}$ is considered normal, otherwise the algorithm is triggered. The excess current $i_{dif}(t)$, as shown in Fig. 5.3, is calculated as the difference between the input current $i(t)$ and the reference current $i_{ref}(t)$ patterns, i.e.

$$i_{dif}(t) = i(t) - i_{ref}(t) \tag{5.1}$$

Preprocessing

Neural network processing requires the input pattern samples to start at the zero crossing of the positive half–cycle. The search for this condition is carried out in the first snapshot. An adjustable number of cycles are

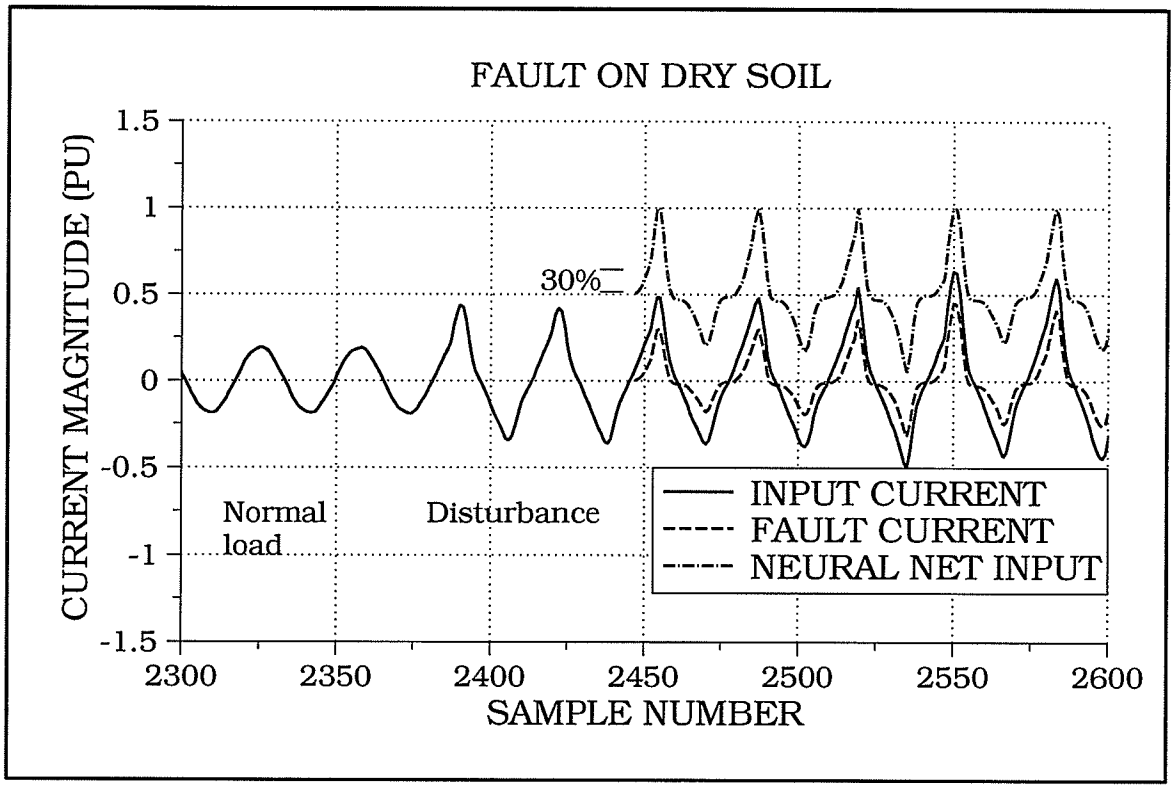

Figure 5.3. Disturbance calculation and neural network preprocessed waveform.

skipped to escape initial high frequency oscillations of other power system events (two cycles were used). On a cycle–to–cycle basis, the sampled values of the difference current are adjusted to analog values between "0" and "1", to suit the network input level: see Fig. 5.3. In the same time the width of current conduction period above 30% of the peak current is determined. The resulting input vectors are passed one–by–one to the artificial neural network (ANN) for pattern identification.

Output

The trained network maps the input to its decision region. The algorithm integrates the ANN "scores" to obtain the detector "output": equation (5.2); this is called output filtering.

$$output_{new} = output_{old} + \frac{\delta t}{\tau} * (score_{new} - output_{old}) \qquad (5.2)$$

where " δt " is the integration time step (1 cycle), and " τ " is the integration time constant (1 second). The detector output is compared with a detection threshold (0.75) to classify whether or not the disturbance is due to an arcing fault. The process is then repeated for the next snapshots. The number of snapshots was limited by the length of test data arrays to 10 sets.

5.3. Results

The arcing fault detection algorithm was tested by four second traces of normal load current disturbed by currents of faults on dry and wet soil, arc welder, computer, fluorescent light, and sinusoidal loads.

Arcing Faults

Training pattern sets, section 4.3.1., supplied to the neural network for faults on dry and wet soil were obtained from the fault current data shown

in Fig. 5.4. This is the reason why the ANN has high scores for these faults. For the fault on dry soil, Fig. 5.4 top, when arcing is almost off near time 3.2 seconds, one would expect a no–fault response; the ANN score went down.

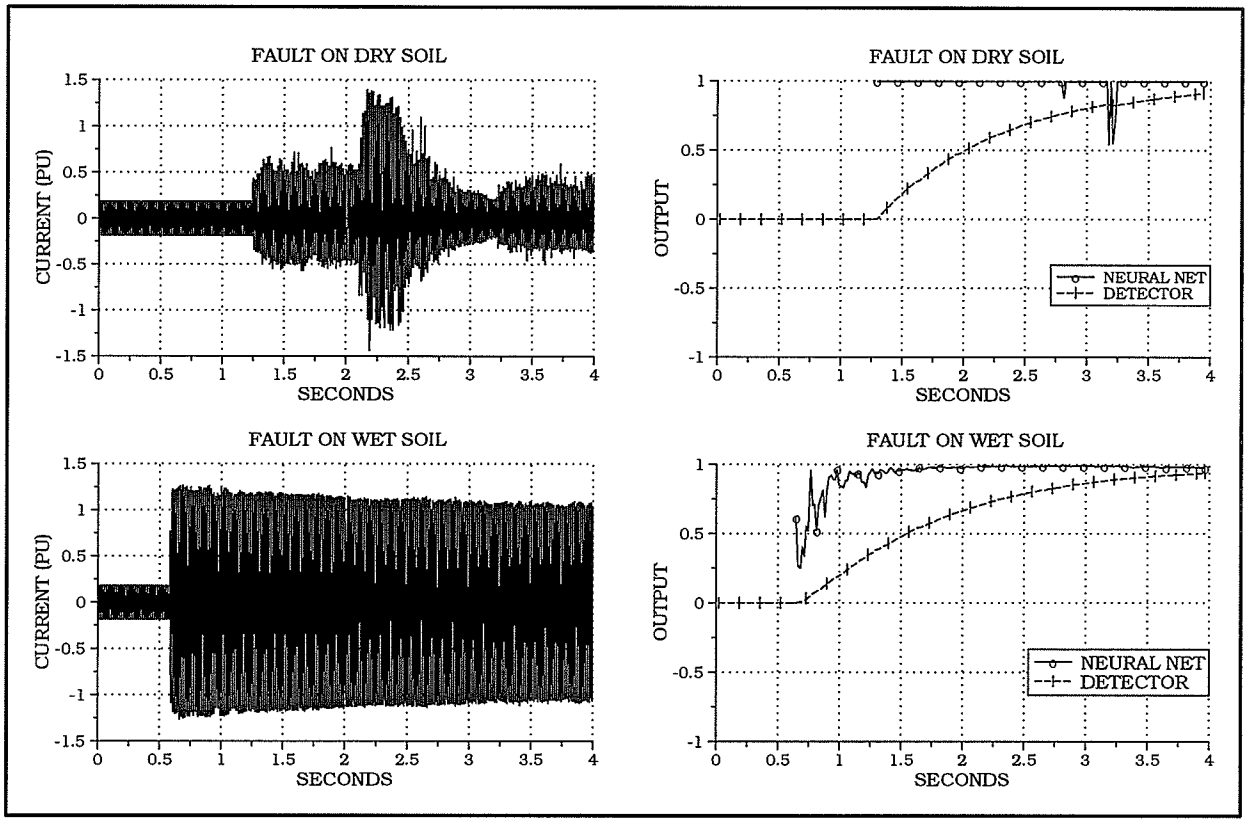

Figure 5.4. Arcing faults on dry (top), and wet soil (bottom). Left: current traces. Right: results.

When the soil is very wet, the fault current pattern is very close to a sinusoid. Therefore, the ANN shows a relatively low score in the first 0.3 seconds of the fault on wet soil: Fig. 5.4 bottom right. As the dissipated energy in the fault bakes the soil, it dries the surface. The current waveform becomes a typical HIF pattern which is easily identified by the ANN.

To examine the *dependability* of the detection algorithm, several fallen conductor faults were staged on dry and wet grassy soil: Figures 5.5 and 5.6 respectively. The collected waveforms contained arc generated noise. In all tests, the results show that the detector output exceeds 75% in less than 4 seconds, indicating a possible fault: see Fig. 5.7.

The results show good promise of applying the neural networks approach in recognizing high impedance fault current patterns. The algorithm needs to be tested for faults on different ground materials. This may require additional training patterns, and could require different ANN architecture or even another network model. Certainly, on–line testing of the algorithm is required. The combination of different test results, and detector hardware would determine the optimum size and number of snapshots, detection threshold level, and decision time required to determine whether the existing disturbance is a permanent or a temporary HIF.

Fault–like and Normal Loads

Arcing loads, such as arc welders and arc furnaces, are loads likely to be confused with high impedance arcing faults. Testing the algorithm with an arc welder load, Fig. 5.8 top, shows how far the above possibility is a problem for the detection algorithm. The detector estimation for this event as a fault was less than 20%. The results of testing the computer, the fluorescent light, and the sinusoidal loads, from second to bottom of Fig. 5.8, indicate that the algorithm will also be *secure* in these events.

Training the ANN with more patterns of non–linear loads that may or may not mimic fault conditions would add to the detector *reliability*.

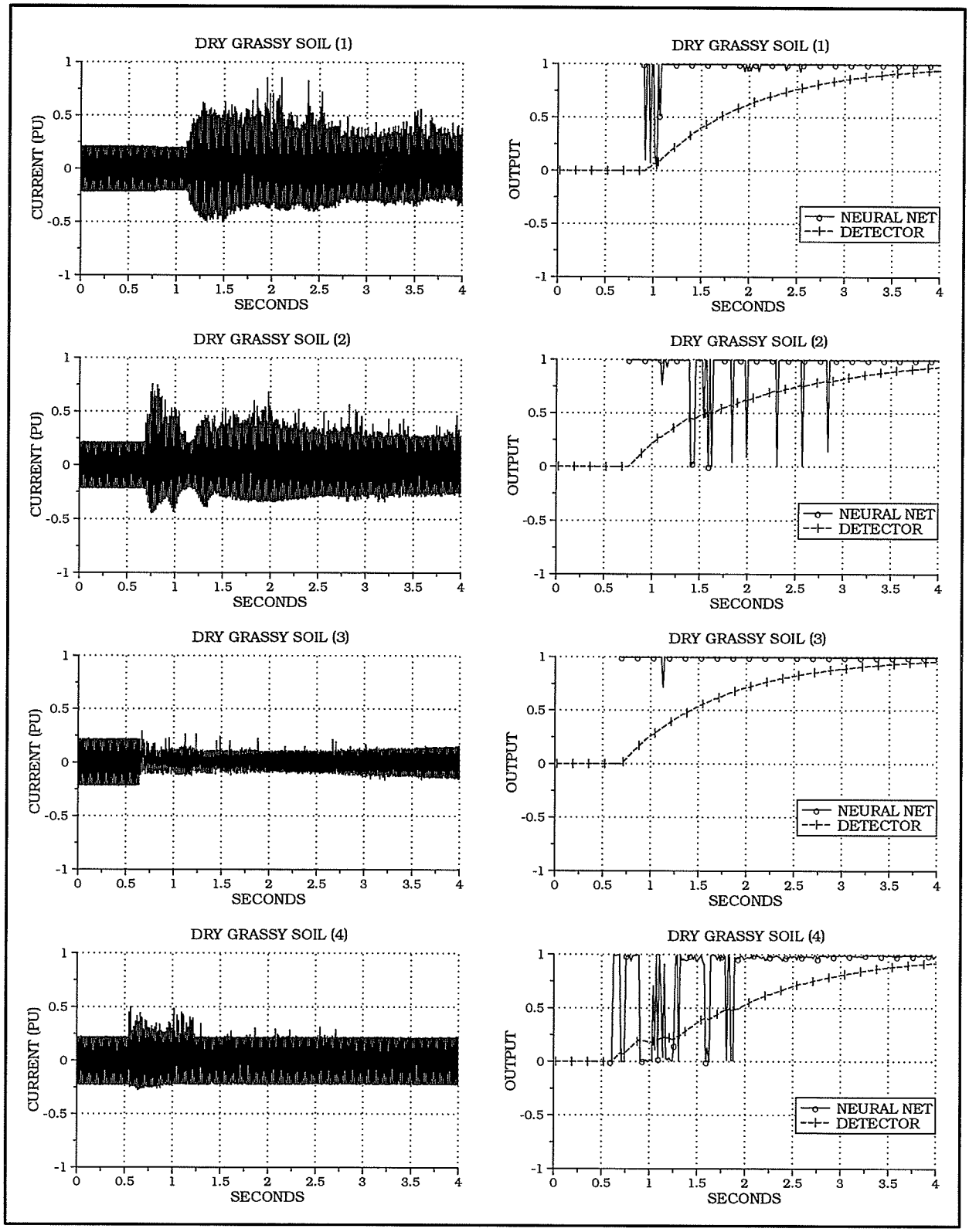

Figure 5.5. Tests of arcing faults on dry grassy soil. Left: current traces. Right: results.

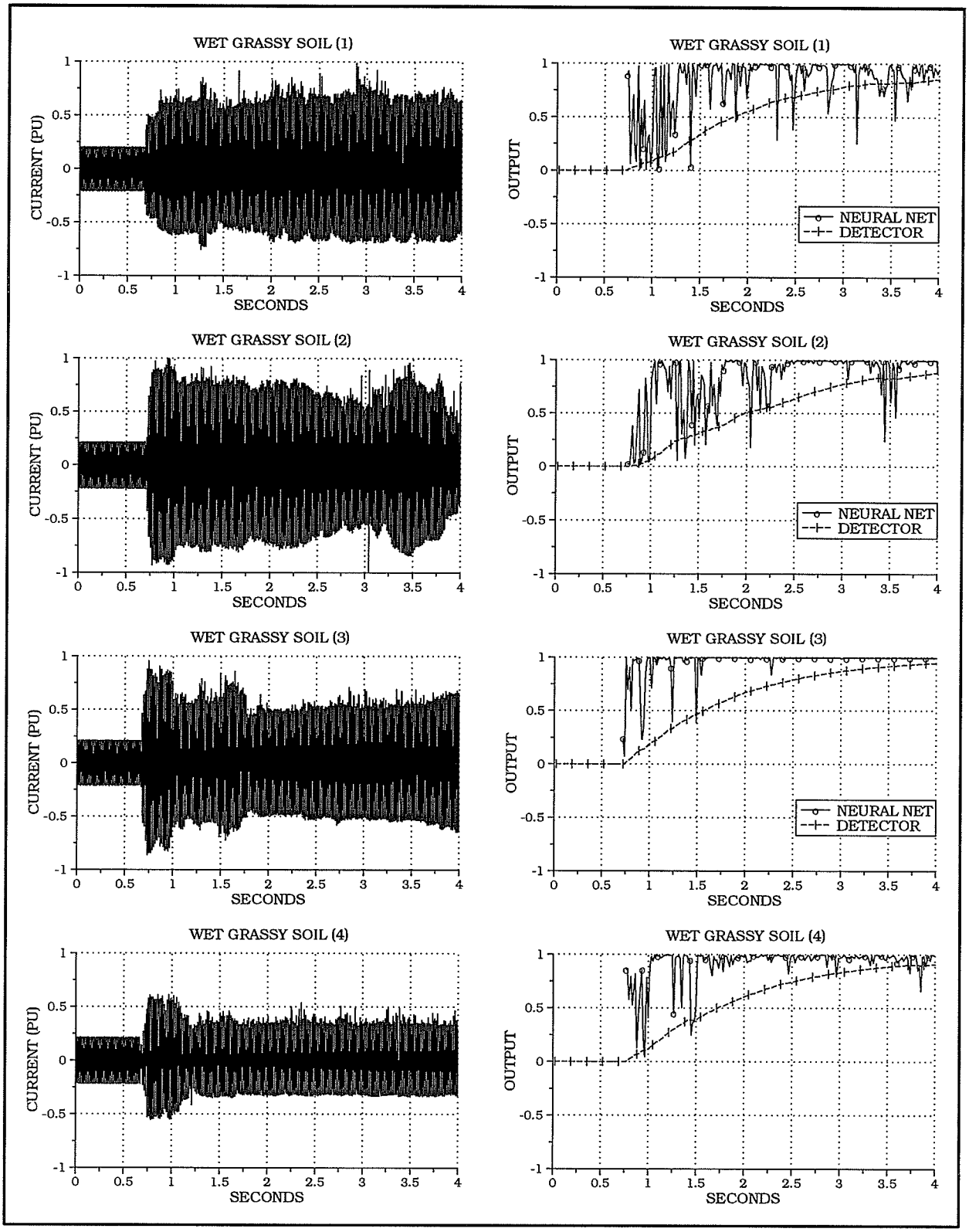

Figure 5.6. Tests of arcing faults on wet grassy soil. Left: current traces. Right: results.

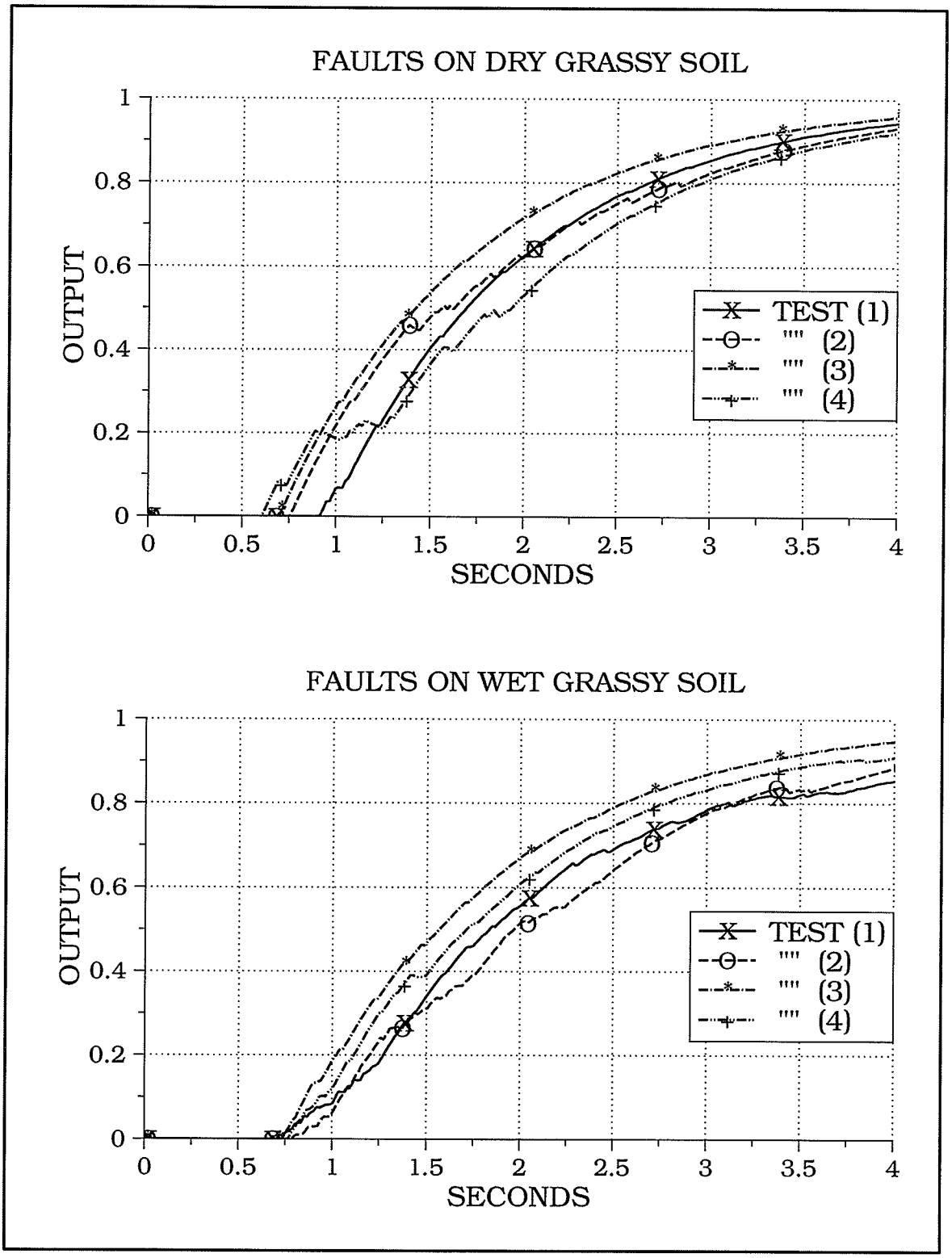

Figure 5.7. Detector output for the arcing fault tests on dry (top) and wet (bottom) grassy soil.

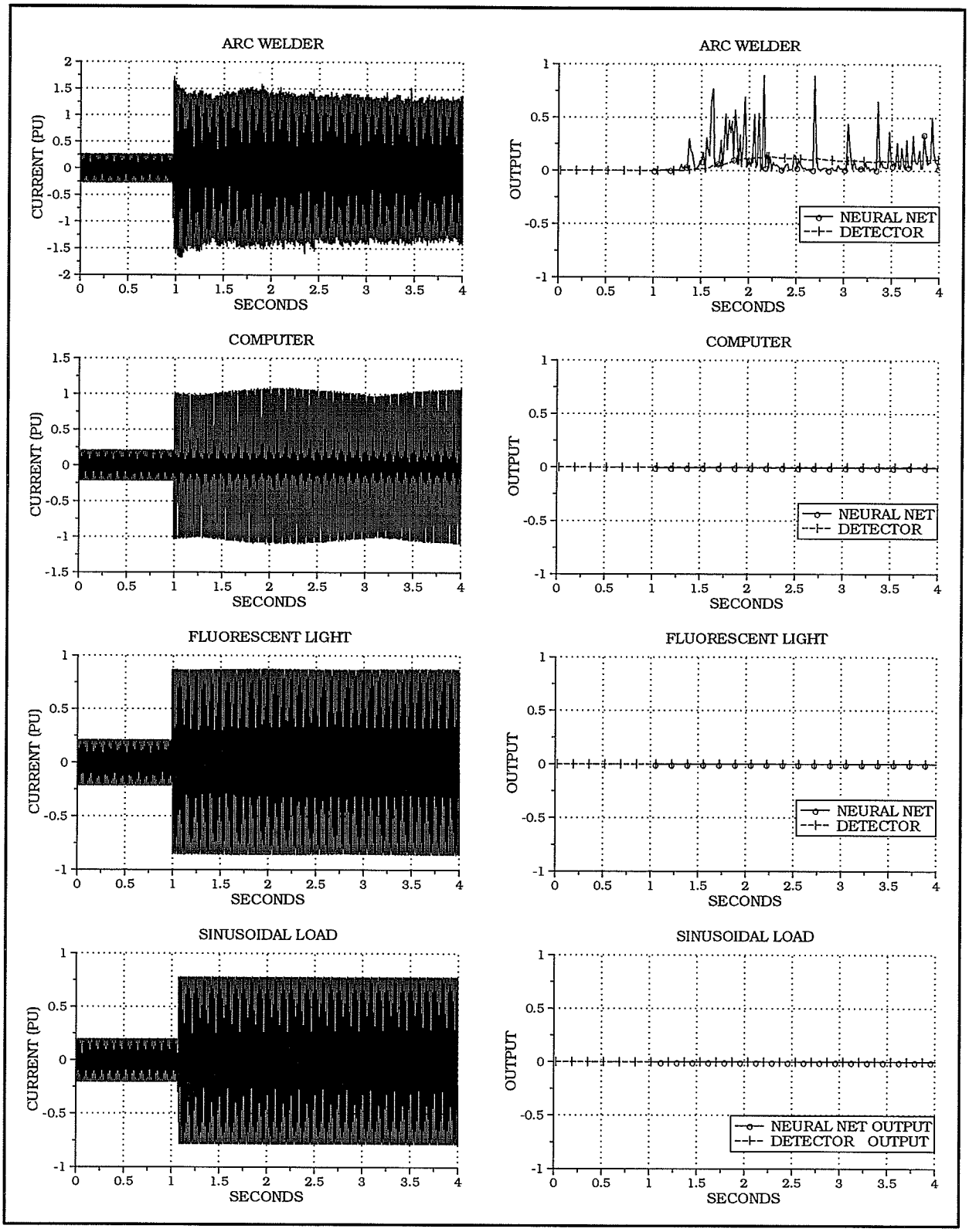

Figure 5.8. Fault–like and normal loads.
Top to bottom: arc welder, computer, fluorescent light, sinusoidal load.

Analysis of Neural Network Performance

Snapshots were selected from Figures 5.5, 5.6 and 5.8 to study the ANN performance as shown in Fig. 5.9. It is seen that the presence of arc noise on top of the positive half–cycle of the current in faults on dry grassy soil is more than its presence on the negative half–cycle: Fig. 5.9 top. This could

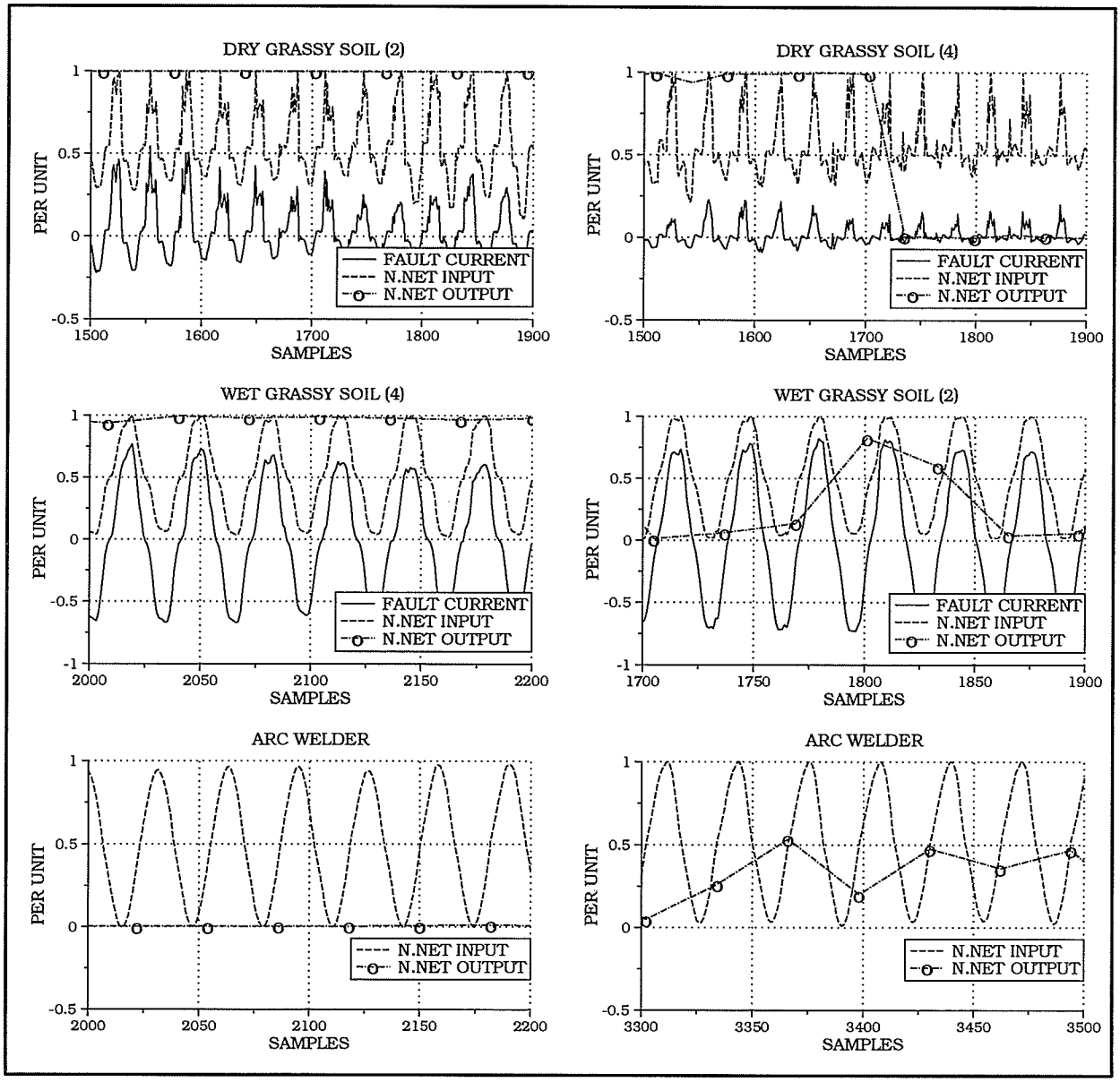

Figure 5.9. Study of the neural net performance on a cycle–to–cycle basis. Left: high performance. Right: low performance.

be due to an absorption ability exhibited by the soil itself. This noise appears when the fault current is small, approximately 1 A, and decreases as the fault current magnitude increases as seen with faults on wet grassy soil: middle of Fig. 5.9. However, the network scores are insignificantly affected by the noise as long as the current waveform takes the typical HIF current pattern. Once the waveform is significantly distorted or comes close to the boundary that separates arcing faults from other events, the ANN would either indicate no–fault condition, top and middle of Fig. 5.9 right, or be confused as seen in bottom of Fig. 5.9 right.

5.4. Selection of the Number of Hidden Nodes and the Learning Mode

The selection of six hidden neurons and pattern mode training, section 4.3.2., was based on the ability of the network to distinguish between HIF, fault–like and sinusoidal loads, and the detector output during conducted tests.

5.4.1. Number of Hidden Nodes

Networks of "4" and "5" hidden neurons have been trained, with the training set described in section 4.3.1., in the pattern learning mode at learning rates of 0.1 and 0.5 respectively. Though the four hidden node network has a good performance in identifying most of the HIF, the arc welder load presented a difficulty to the network: Fig. 5.10. The five hidden nodes network did have a good response in all tests; however, when tested with HIF on wet grassy soil test (2), Fig. 5.11, the performance of both the "4" and "5" hidden node networks was low compared to the used "6" hidden node network. The selection of six hidden nodes was based on these results.

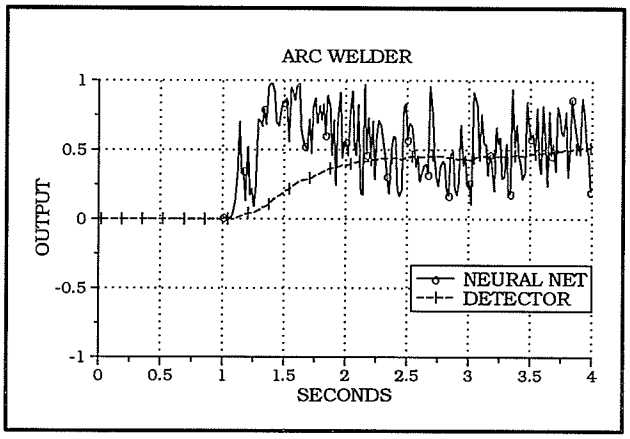

Figure 5.10. Performance of a four hidden neuron ANN to arc welder load: pattern learning mode, η = 0.1.

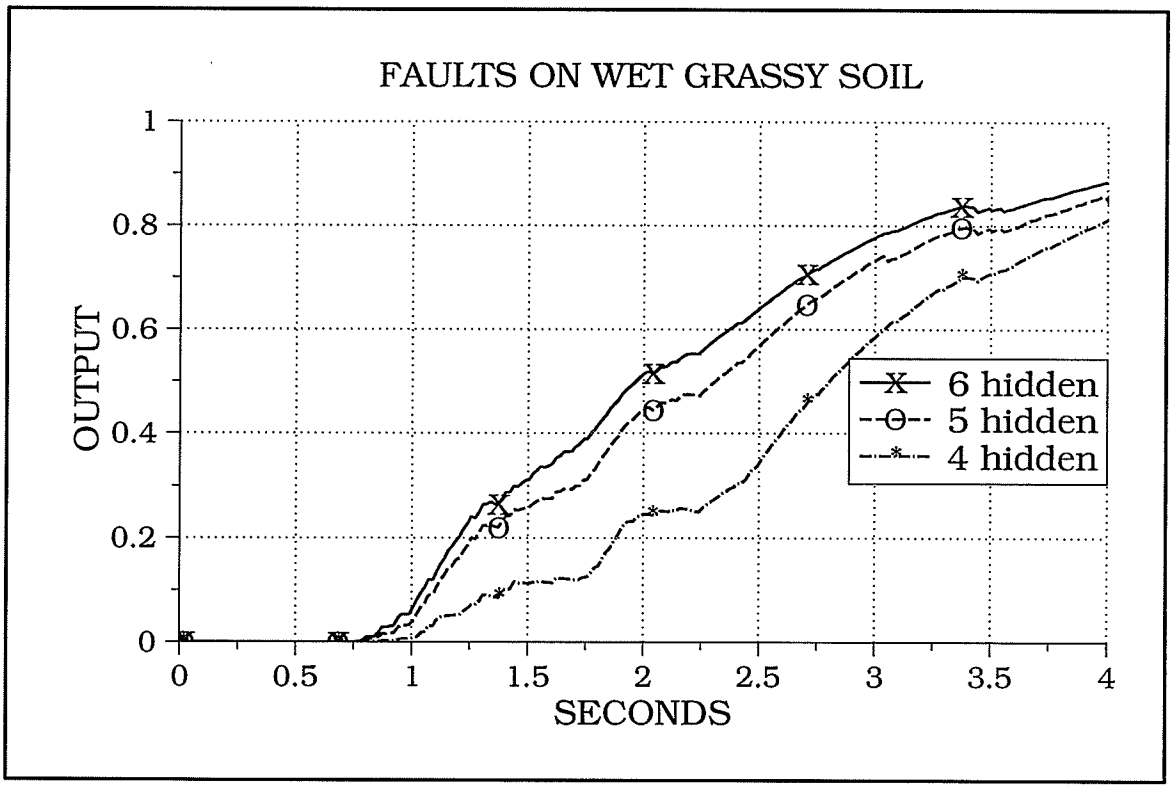

Figure 5.11. Study of the neural net performance using different number of hidden nodes. Six and five hidden nodes η = 0.5, four hidden nodes η = 0.1.

5.4.2. Learning Mode

The epoch learning mode was used to train the six hidden neuron network at a learning rate of 0.25. The network suffered the same drawbacks

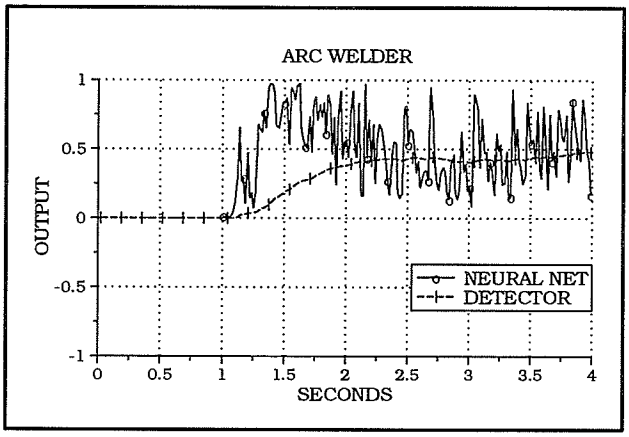

Figure 5.12. Performance of a six hidden neuron ANN to arc welder load: epoch learning mode, η = 0.25.

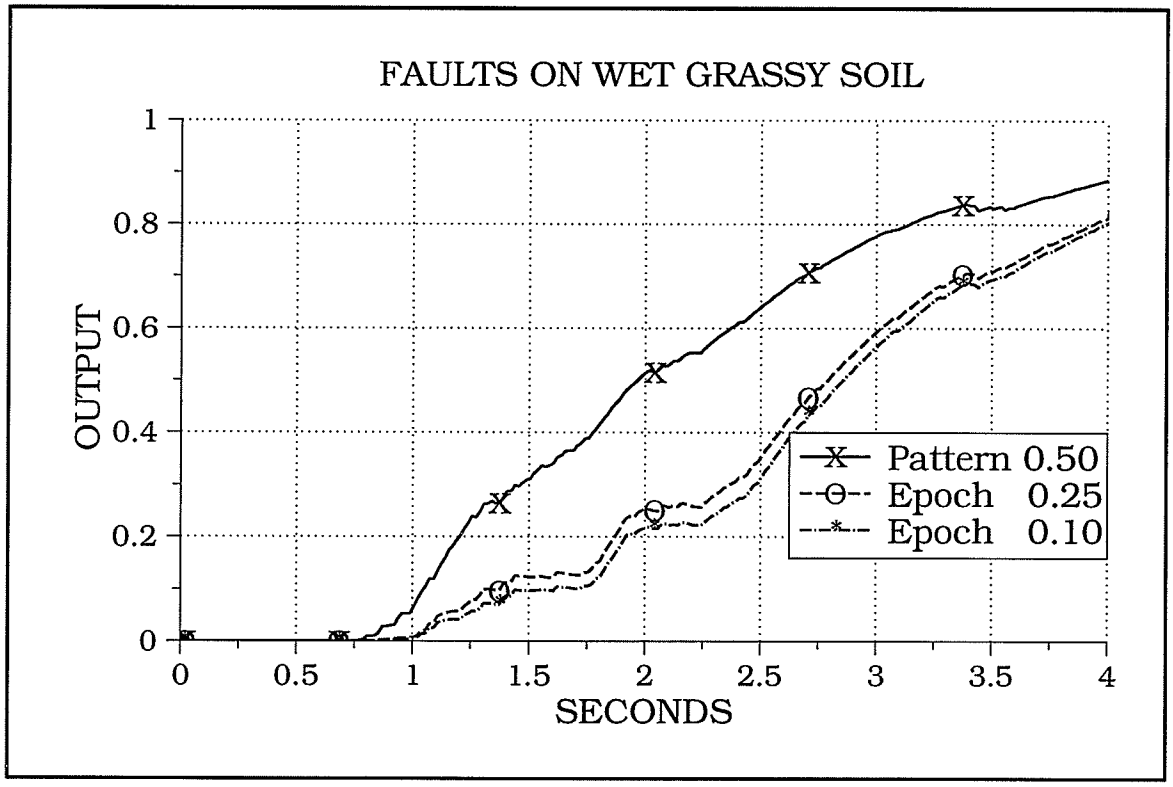

Figure 5.13. Study of the neural net performance using six hidden nodes at different learning modes and rates.

as the four hidden nodes network: Fig. 5.12. It was thought that the network was stuck in a local minimum. The learning rate η was decreased to 0.1 in hope to improve the performance; however, the results became worse: Fig. 5.13. Learning by pattern was therefore used instead.

Conclusions

Field tests on the Manitoba Hydro system have confirmed the levels and waveshapes associated with downed conductor faults.

An arcing high impedance fault laboratory model was validated in the high voltage laboratory at the University of Manitoba. The laboratory fault current proved to be a credible source of data acquisition for use in solving the arcing faults detection problem.

Existing detection methods have limitations as to the type of high impedance fault that can be detected; furthermore, there are loads that mimic fault currents, e.g. arcing loads. Some existing detection methods can fail to distinguish between faults and fault–like loads. Fault–like loads that were investigated included arc welders, computers, and fluorescent lamps. Thus the *security* (relay's ability to **not** trip when it shouldn't) of a new detection method could be tested.

A high impedance arcing faults detection algorithm was designed, using an artificial neural network (ANN). The algorithm was tested by signals consisting of normal load current disturbed by currents of faults on dry and wet grassy soil, arc welder, computer, fluorescent light, and sinusoidal loads. The detector was successful in separating arcing faults from arcing and nor-

mal loads. The detector performance was outstanding even under noisy signal conditions.

Future Work

A *reliable* high impedance fault (HIF) detector is a relay that trips only under the existence of a HIF. The relay *reliability* is a compromise between *dependability* (relay's ability to trip when it should) and *security* (relay's ability to **not** trip when it shouldn't). The unpredictable behavior of high impedance fault current suggests the use of other detection methods to work in parallel with the proposed algorithm to increase the relay reliability. The variation in current magnitude each cycle (randomness), and from one half–cycle to the other (asymmetry/flicker) could be used as features input to an artificial neural network (ANN), or in a detection algorithm to indicate arcing.

On–line testing is required to confirm the algorithm applicability. The outcome from algorithm implementation could demand:

- a different neural network architecture (e.g. different number of hidden neurons),
- more training patterns of faults on different ground materials and surface conditions as well as fault–like and normal loads,
- a different learning procedure (learning rate, mode, or degree of neuron transfer function non–linearity), and

– the suitable values of detection thresholds.

The ANN approach is not only promising in high impedance fault detection but also for abnormal events identification and classification, though a different neural network model may be required.

Bibliography

[1] *Downed Power Lines: Why they Can't Always be Detected.* A publication of the IEEE/PES, February 1989.

[2] *Detection of Down Conductors on Utility Distribution Systems.* IEEE Tutorial Course no. 90EH0310–3–PWR, IEEE/PES 1990 Summer Meeting, July 15–19, 1990.

[3] B.M. Aucoin, B.D. Russell, C.L. Benner. *High Impedance Fault Detection for Industrial Power Systems.* IEEE Industrial Application Society Conference, San Diego, October, 1989.

[4] D.I Jeerings, J.R. Linders. *Ground Resistance Revisited.* IEEE Transactions on Power Delivery, Vol. PWRD–4, No. 2, April 1989, pp. 949–956.

[5] J. Carr. *Detection of High Impedance Faults on Multi–Grounded Primary Distribution System.* IEEE Transactions on Power Apparatus and Systems, Vol. PAS–100, No. 4, April 1981, pp. 2008–2016.

[6] H. Calhoun, M.T. Bishop, C.H. Eichler, R.E. Lee. *Development and Testing of an Electromechanical Relay to Detect Fallen Distribution Conductors.* IEEE Transactions on Power Apparatus and Systems, Vol. PAS–101, No. 6, June 1982, pp. 1643–1650.

[7] B.M. Aucoin, B.D. Russell. *Distribution High Impedance Fault Detection Utilizing High Frequency Current Components.* IEEE Transactions on Power Apparatus and Systems, Vol. PAS–101, No. 6, June 1982, pp. 1596–1606.

[8] B.M. Aucoin, J. Zeigler, B.D. Russell. *Feeder Protection and Monitoring System, Part I: Design, Implementation and Testing.* IEEE Transactions on PAS, Vol. PAS–104, No. 4, April 1985, pp. 873–880.

[9] B.M. Aucoin, J. Zeigler, B.D. Russell. *Feeder Protection and Monitoring System, Part II: Staged Fault Test Demonstration.* IEEE Transactions on PAS, Vol. PAS–104, No. 6, June 1985, pp. 1456–1462.

[10] B.M. Aucoin, B.D. Russell. *Detection of Distribution High Impedance Faults Using Burst Noise Signals Near 60 Hz.* IEEE Transactions on Power Delivery, Vol. PWRD–2, No. 2, April 1987, pp. 342–348.

[11] C.L. Benner, P.W. Carswell, B.D. Russell. *Improved Algorithm for Detecting Arcing Faults using Random Fault Behavior.* Southern Electric Industry Application Symposium, New Orleans, LA, Nov. 15–16, 1988.

[12] B.D. Russell, R.P. Chinchali. *A Digital Signal Processing Algorithm for Detecting Arcing Faults on Power Distribution Feeders.* IEEE Transactions on Power Delivery, Vol. PWRD–4, No. 1, January 1989, pp. 132–140.

[13] W.H. Kwon, et al. *High Impedance Fault Detection Utilizing Incremental Variance of Normalized Even Order Harmonic Power.* Paper no. 90 SM

349–1 PWRD, presented at IEEE/PES 1990 Summer Meeting, July 15–19, 1990.

[14] A.F. Sultan, G.W. Swift. *Discussion: High Impedance Fault Detection Utilizing Incremental Variance of Normalized Even Order Harmonic Power.* IEEE Transactions on Power Delivery, Vol. PWRD–6, No. 2, April 1991. pp. 564.

[15] S. Ebron, D.L. Lubkeman, M. White. *A Neural Network Approach to The Detection of Incipient Faults on Power Distribution Feeders.* IEEE Transactions on Power Delivery, Vol. PWRD–5, No. 2, April 1990, pp. 905–914.

[16] S. Ebron. *A Neural Network Processing Strategy for the Detection of High Impedance Faults.* Master's Thesis, Electrical and Computer Engineering Department, NCSU, 1988.

[17] *High Impedance Fault Detection Using Third Harmonic Current.* EPRI Report EL–2430, prepared by Hughes Aircraft Co., June 1982.

[18] D.I Jeerings, J.R. Linders. *Unique Aspects of Distribution System Harmonics Due to High Impedance Ground Faults.* IEEE Transactions on Power Delivery, Vol. PWRD–5, No. 2, April 1990, pp. 1086–1094.

[19] D.I Jeerings, J.R. Linders. *Discussion: IEEE Tutorial Course; Detection of Downed Conductors on Utility Distribution Systems.* IEEE/PES Winter Meeting, Atlanta, GA, February 8, 1990.

[20] D.I. Jeerings, J.R. Linders. *A Practical Protective Relay for Down–conductor Faults.* Paper no. 90 SM 333–5 PWRD, presented at IEEE/PES 1990 Summer Meeting, July 15–19, 1990.

[21] *High Impedance Fault Test Conducted by Hydro and University Engineers.* Hydrogram, Vol. 19, No. 27, October 5, 1989.

[22] *Fault Induced Wave Distortion of Interest to Relay Engineers.* A Report prepared by the PSRC. IEEE Transactions on PAS, Vol. PAS–104, No. 12, December 1985, pp. 3574–3584.

[23] A.E. Emanuel, D.Cyganski, J.A. Orr, S. Shiller, E.M. Gulachenski, *High Impedance Fault Arcing on Sandy Soil in 15 kV Distribution Feeders: Contribution to the Evaluation of Low Frequency Spectrum.* IEEE Transactions on Power Delivery, Vol. PWRD–5, No. 2, April 1990, pp. 676–686.

[24] Reinhold Rüdenberg. *Transient Performance of Electric Power Systems: Phenomena in Lumped Networks.* MIT Press, 1969.

[25] A.F. Sultan, G.W. Swift. *Security Testing of High Impedance Fault Detectors.* IEEE/WESCANEX May 29 & 30 1991, Regina, Saskatchewan, CANADA.

[26] *MathCAD™ Version 2.5 User's Guide.* MathSoft, Inc., 1986–1989.

[27] *DARPA Neural Network Study.* AFCEA International Press, Fairfax, Virginia, November 1988.

[28] C.R. Parten et al. *Handbook of Neural Computing Applications.* Academic Press, Inc., 1990.

[29] S.C. Huang, Y.F. Huang. *Bounds on Number of Hidden Neurons in Multi–Layer Perceptrons.* IEEE Transactions on Neural Networks, Vol. 2, No.1, January 1991, pp. 47–55.

[30] M.A. Sartori, P.J. Antsaklis. *A Simple Method to drive Bounds on the Size and to train Multi-Layer Neural Network.* IEEE Transactions on Neural Networks, Vol. 2, No.4, July 1991, pp. 467–471.

[31] Rumelhart, McClelland, and the PDP Research Group. *Parallel Distributed Processing: Exploration in the Microstructure of Cognition. Volume 1: Foundations.* MIT Press, 1989.

[32] R.P. Lippman. *An Introduction to Computing with Neural Nets.* IEEE ASSP Magazine, April 1987.

[33] McClelland and Rumelhart. *Explorations In Parallel Distributed Processing.* MIT Press, 1988.

[34] *Neural Networks Capabilities and Applications.* IEEE Video–conference, Seminar Via Satellite, September 27, 1989.

[35] A.F. Sultan, G.W. Swift, D.J. Fedirchuk. *Detection of High Impedance Arcing Faults Using a Multi-Layer Perceptron.* IEEE/PES Winter Meeting, New York, N.Y., Jan. 1992.

[36] *Some Considerations for the AMPS System.* The University of Manitoba Power Systems Group, AMPS project guidelines, G.W. Swift, June 1991.

A Detection Algorithm: Program List

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>

                                        /* new.c                    May 4 91*/

/* PROGRAM TO EXTRACT THE FEATURES OF THE LINE CURRENT WAVEFORM */

main()

{
      menu ();
again:  printf ("\n");
                options ();

                switch(ch[0]) {
                        case '1':
                                read_oct ();
                                time=SEC_PER_SAMPLE;
                                printf ("time=%f \n",time);
                                write_data ();
                                break;
                        case '2':
                                read_oct ();
                                time=SEC_PER_CYCLE;
                                all ();
                                break;
                        case '3':
                                read_oct ();
                                time=SEC_PER_CYCLE;
                                random ();
                                break;
                        case '4':
                                read_dat ();
                                time=SEC_PER_SAMPLE;
                                write_data ();
                                time=1.0;
                                write_mbplot ();
                                time=32.0;
                                all ();
                                break;
                        case '5':
                                read_dat ();
                                time=SEC_PER_CYCLE;
                                random ();
                                break;
                        case '6':
                                read_lotus_2 ();
                                process_lotus_2 (record_1,array_1);
                                time=SEC_PER_SAMPLE;
                                write_data_2 ();
                                random ();
                                 break;
                        case '7':
                                read_oct ();
                                time=SEC_PER_SAMPLE;
                                break;
                                case '8':
                                read_dat ();
                                time=SEC_PER_SAMPLE;
                                test ();
                                break;
                        case '9':
                                time=SEC_PER_SAMPLE;
                                read_lotus_out ();
                                test ();
                                break;
                        default:
                                exit ();
                        }
                        printf ("..... DONE .....");
                        printf ("\n");
                        goto again;
}
```

```
                /* display                                  May 2 91 */

menu ()
    {
     printf (" ------------------------------ \n");
     printf ("|          MENU                | \n");
     printf (" ------------------------------ \n");
     printf ("|   1. Extract Features.       |\n");
     printf (" ------------------------------ \n");
     printf ("|   2. Exit.                   |\n");
     printf (" ------------------------------ \n");
     printf ("    Enter Your Choice: ");
     scanf ("%4s",ch);
     printf ("\n");
                if (ch[0] == '2')
                {
     printf ("********** THE END ********** \n");
     exit();
                }
     else
                return;
     }

options ()
                {
      printf (" ------------------------------------------------------------ \n");
     printf ("|                    OPTIONS                    | \n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   1. Transfere an OCTAL file to a data file.          |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   2. Extract dc, ac, and randomness starting from OCTAL file.  |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   3. Calculate RANDOMNESS starting from OCTAL file.          |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   4. Extract dc, ac, and randomness starting from data file.  |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   5. Calculate RANDOMNESS starting from data file.          |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   6. Process a LOTUS-123 file.                        |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   7. Test current change in an OCTAL file.            |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   8. Test current change in an data file.             |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   9. Test current change in an LOTUS file.            |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("|   e. End.                                             |\n");
      printf (" ------------------------------------------------------------ \n");
     printf ("    Enter Your Choice: ");
     scanf ("%4s",ch);
     printf ("\n");
                if (ch[0] == 'e')
     {
     printf ("********** THE END ********** \n");
     exit();
                }
                else
     return;
                }
```

```
/* constant                                    July 8 91 */

#define ARRAY_SIZE 81920
#define SAMPLES_PER_CYCLE 32
#define BLOCK_SIZE 2048
#define ARRAY_SIZE_2  2*ARRAY_SIZE
#define BLOCK_SIZE_2 2*BLOCK_SIZE
#define ARRAY_SIZE_CYCLES  ARRAY_SIZE/SAMPLES_PER_CYCLE
#define JUNK 2056
#define NULL 0
#define STRING_LENGTH 30
#define PI 3.14159265359
#define TWO_PI 2.0*PI
#define TIME_STEP TWO_PI/(float)SAMPLES_PER_CYCLE
#define SQUARE(x) (x)*(x)
#define FREQUENCY 60.0
#define SEC_PER_CYCLE    1.0/FREQUENCY
#define SEC_PER_SAMPLE SEC_PER_CYCLE/(float)SAMPLES_PER_CYCLE
#define CHECK_CYCLES 20
#define TIME_C SEC_PER_CYCLE
#define TIME_S SEC_PER_SAMPLE
#define SECONDS ((float) CHECK_CYCLES)*SEC_PER_CYCLE
#define CHECK_SAMPLES CHECK_CYCLES*SAMPLES_PER_CYCLE
#define TOTAL_1 ARRAY_SIZE/CHECK_SAMPLES
#define TOTAL CHECK_CYCLES*SAMPLES_PER_CYCLE
#define THRESHOLD 0.01
#define NOISE 0.1
#define DC 0.0
#define LAST_SET 20
#define WEI 500
#define LEVEL 0.3
#define INPUT SAMPLES_PER_CYCLE+1
```

```
                /* variables                                    July 8 91 */

/* PROGRAM TO EXTRACT THE FEATURES OF THE LINE CURRENT WAVEFORM */

    FILE *in_f, *out_f, *fopen (), *fclose ();

    char name[STRING_LENGTH],in_name[STRING_LENGTH],out_name[STRING_LENGTH];
                char str[STRING_LENGTH];

    char          in[]=(".in");
    char          rec[]=(".rec");
    char          hex[]=(".hex");
    char          prn[]=(".prn");
    char          oct[]=(".oct");
    char          dat[]=(".zdat");
    char          dcf[]=(".zdc");
    char          acf[]=(".zac");
    char          raf[]=(".zra");
    char          rmsf[]=(".zrms");
    char          diff[]=(".zdif");
    char dc_diff[]=(".zdif_dc");
    char ac_diff[]=(".zdif_ac");
    char ra_diff[]=(".zdif_ra");
    char          rndf[]=(".zrnd");
    char dc_rndf[]=(".zrnd_dc");
    char ac_rndf[]=(".zrnd_ac");
    char ra_rndf[]=(".zrnd_ra");
    char ch[1];

    int j,jj;                                          /*index of sample number*/
    int Po;                    /*index of first data line*/
    int Pn,Pnn;                                        /*index of last data line*/
    int c;                    /*index of cycle number*/
    int Co;                    /*index of first cycle*/
    int Cn;                    /*index of last cycle*/
    int car;                   /*character to indicate EOF*/
    int row,column,rows,columns,step,last_record,target;
    int N[6]={ 4, 8, 16, 32, 64, 128};
     int T[32]={0,1,2,3,4,5,6,7,8,7,6,5,4,3,2,1,0,-1,-2,-3,-4,-5,-6,-7,-8,-7,-6,-5,-4,-3,-2,-1};

    int quantized_data[ARRAY_SIZE_2];

                float time,rms_1,rms_2,rms_3,r,a,b,m;
    float record_1[ARRAY_SIZE],array_1[ARRAY_SIZE];
    float record_2[ARRAY_SIZE],array_2[ARRAY_SIZE];
                float store_1[SAMPLES_PER_CYCLE],store_2[SAMPLES_PER_CYCLE];
                float store_3[2*SAMPLES_PER_CYCLE];

                int width[ARRAY_SIZE/SAMPLES_PER_CYCLE];

    float dc[ARRAY_SIZE_CYCLES];    /*average value per cycle*/
    float ac[ARRAY_SIZE_CYCLES];    /*rms value per cycle*/
    float ra[ARRAY_SIZE_CYCLES];    /*rms value per cycle*/
    float dc_dif[ARRAY_SIZE_CYCLES];
    float ac_dif[ARRAY_SIZE_CYCLES];
    float dc_rnd[ARRAY_SIZE_CYCLES];
    float ac_rnd[ARRAY_SIZE_CYCLES];

    float seg[ARRAY_SIZE_CYCLES];
    float dif[ARRAY_SIZE_CYCLES];

    double record_3[CHECK_SAMPLES],array_3[CHECK_SAMPLES];
```

```
                    /* calculator                    May 15 91 */

float absolute (a)
        float a;
        {
        float b;
        if ((a>0.0) | |(a==0.0))
        b=a;
        else
        b=-a;
        return b;
        }

top (a,b)
        int a,b;
        {
        int result,c;
        c=a%b;
        if (c == 0)
        result=a/b;
        else
        result=a/b+1;
return (result);
        }

difference (array,result)
        float array[],result[];
        {
                        float *p,*p_end;
                        int y;
        Co=0;
        for (column=0;column<columns;++column)
        {
        step=Cn*column;
        for (c=Co;c<Cn;++c)
        {
                        y=step+c+1;
                        p=&array[y-1];
                        p_end=&array[y];
        result[y]=*p_end-*p;
        }
        }
        }

integrate (array,result)
        float array[ARRAY_SIZE],result[ARRAY_SIZE];
        {
                        float *p,*p_end,*p2,*p2_end,tc=0.00333;
                        int y;
                        for (column=0;column<columns;++column)
        {
        step=Cn*column;
        result[step]=0.0;
        for (c=Co;c<Cn;++c)
        {
                        y=step+c+1;
                        p_end=&array[y];
                        p=p_end-1;
        result[y]=result[y-1]+tc*(*p_end - result[y-1]);
                        if (result[y] > 900)
                        result[y]=999.0;
        }
        }
        }

dc_ac (record,dc,ac,ra)
        float record[];
        float dc[],ac[],ra[];
        {
        /* Evaluate the "dc", "ac", and "ra" values of the wave*/
        int column,y;
        float *p,*p_end;
        float *pointer, *pointer_end;
        Co=0;
        j=0;
        for (column=0;column<columns;++column)
        {
        step=rows*column;
        for (c=Co;c<Cn;++c)
        {
                        y=step+c*SAMPLES_PER_CYCLE;
                        p=&record[y];
                        p_end=p+SAMPLES_PER_CYCLE;
        dc[j]=0.0;
        ac[j]=0.0;
        ra[j]=0.0;
                for (;p<p_end;++p)
                {
                dc[j]+=*p;
                ac[j]+=SQUARE (*p);
                }
        dc[j]= dc[j]/SAMPLES_PER_CYCLE;
        ac[j]= ac[j]/SAMPLES_PER_CYCLE;
        ac[j]= sqrt(ac[j]);
        if (ac[j] > 1.0)
        ra[j]=dc[j]/ac[j];
        ++j;
        }
        }
        }
```

```
segma (array,seg)
/* needs some adjustment */
        float array[],seg[];
        {
                        int l=0,k=0,column,y,Co=0;
                        float x,*p,*p_end;
                        for (column=0;column<columns;++column)
                        {
                        step=Cn*column;
        k=N[l]+step;
        for (c=Co+N[l];c<=Cn;++c)
        {
        y=step+c;
        p=&array[y-N[l]];
        p_end=&array[y];
        seg[k]=0.0;
                for (;p<p_end;)
                {
                x=*p++;
                seg[k]+=x*x;
                }
        seg[k]=sqrt(seg[k]/N[l]);
        ++k;
        }
                        k-=1;
        }
        }

oct_decimal (l,array)
        int l[];
        float array[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int *q_pointer,*p1,*p2,*p3,*p4,count1,count2,k;
        float *pointer, *pointer_end;

                        q_pointer=(int *) calloc(ARRAY_SIZE_2,sizeof(int));
        if ((q_pointer == NULL))
        {
        printf ("Memory allocation is not successful.\n");
        return;
        }

        j=0;
        count1=0;
        pointer=array;
                        q_pointer=l;
                        p4=q_pointer;
                        p1=p4;
start:                  p2=p1+BLOCK_SIZE_2-1;
                        for (;p1<p2;j++,p4++)
                        {
                        if ((j == ARRAY_SIZE) || ((car=getc(in_f)) == EOF))
                        goto end;
                        p3=p1+1;
                        fscanf (in_f,"%o %o",p1,p3);
                        *p4=(*p1)*0400+(*p3);
                        p1+=2;
                        }
                        ++count1;
                        p2=p1+7;
        for (;p1<p2;)
        {
        if ((j == ARRAY_SIZE) || ((car=getc(in_f)) == EOF))
        goto end;
        fscanf (in_f,"%o",p1);
        p1+=1;
        }
                        goto start;
end:    Po=0;
        k=(count1)*BLOCK_SIZE;
        Pn=j-1;
        last_record=(k/BLOCK_SIZE);
printf ("j=%7d , Pn=%7d , last_record=%7d\n",j,Pn,last_record);
                        for (j=0;j<Pn;++j)
        {
        if ((i[j] & 04000) != 0)
        {
        i[j]=~(i[j] | 037777774000);
        i[j]=-(i[j]+1);
        }
        else
        i[j]=i[j] & 07777;
        *pointer=i[j]*264/1024;
        ++pointer;
                        }
                        free (q_pointer);
        }

hex_decimal (l,array)
        int l[];
        float array[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int *p1,*p2,*p3,count1,count2,k=10;
        float *pointer, *pointer_end;
        j=0;
        count1=0;
        pointer=array;
                        count2=1;
```

```
                    p1=i;
                    p2=p1+k;
start:              for (;p1<p2;++p1)
        {
        if ((car=getc(in_f)) == EOF)
        goto end;
        fscanf (in_f,"%x",p1);
                    }
                    p1-=k;
                    p2=p1+k;
start2:             for (count1=0;p1<p2;++p1)
                    {
        if (((*p1) & 0xf000) == 0)
        ++count1;
                    }
                    if (count1 == k)
                    {
                    p1-=k;
                    for (;p1<p2;++p1,++j)
                    i[j]=*p1;
                    goto read;
                    }
                    ++count2;
        if ((car=getc(in_f)) == EOF)
        goto end;
                    ++p2;
                    fscanf (in_f,"%x",p1);
                    p1=p2-k;
                    goto start2;
read:               p3=p2+BLOCK_SIZE-k;
                    for (;p1<p3;++j,++p1)
                    {
                    if ((j == ARRAY_SIZE) || ((car=getc(in_f)) == EOF))
                    goto end;
                    fscanf (in_f,"%x",p1);
                    i[j]=*p1;
                    }
                    p2=p1+k;
                    goto start;
end:    Po=0;
        Pn=j;
        last_record=(Pn/BLOCK_SIZE);
printf ("j=%7d , Pn=%7d , last_record=%7d\n",j,Pn,last_record);
                    for (j=0;j<Pn;++j)
        {
        if ((i[j] & 0x0800) != 0)
        {
        i[j]=~(i[j] | 0xfffff800);
        i[j]=-(i[j]+1);
        }
        else
        i[j]=i[j] & 0x0fff;
        *pointer=i[j]*264/1024;
        ++pointer;
                    }
        }


random_detect (array,result)
        float array[],result[];
        {
                    float *p,*p_end,*p2,*p2_end,temp[ARRAY_SIZE],tc=0.001667;
                    int y;
        Co=0;
        for (column=0;column<columns;++column)
        {
        step=Cn*column;
        for (c=Co;c<Cn;++c)
        {
                    y=step+c;
                    p=&array[y];
        if (*p < 0.0)
        temp[y]=-*p;
        else
        temp[y]=*p;
        }
        }
                    for (column=0;column<columns;++column)
        {
        step=Cn*column;
        result[step]=0.0;
        for (c=Co;c<Cn;++c)
        {
                    y=step+c+1;
                    p_end=&temp[y];
                    p=p_end-1;
        result[y]=result[y-1]+tc*(*p_end-result[y-1]);
                    if (result[y] > 900)
                    result[y]=999.0;
        }
        }
        }

process_lotus (array,record)
                    int array[];
                    float record[];
                    {
                    int *pointer,*pointer_end;
                    float *p;
                    pointer=array;
                    p=record;
                    pointer_end=pointer+Pn;
```

```
        printf ("Enter offset : ");
        scanf ("%d",&target);
        printf ("\n");

                        for (;pointer<pointer_end;++pointer,++p)
                        *p=((float) ((*pointer)-target))/800.0;
                        }


process_lotus_2 (array,record)
        float array[];
        float record[];
        {
        float *pointer,*pointer_end;
        float *p;
        pointer=array;
        p=record;
        pointer_end=pointer+Pn;

        printf ("Enter factor : ");
        scanf ("%d",&target);
        printf ("\n");

        for (;pointer<pointer_end;++pointer,++p)
        *p=(*pointer)/((float) target);
        }

feature(disturb,avg,rms)
        double disturb[];
                        float avg[],rms[];
        {
        double *p,*p_end;
        float *pa,*pr;
        p=disturb;
        p_end=p+SAMPLES_PER_CYCLE;
        pa=avg;
        pr=rms;
                        Co=0;
                        for (c=Co;c<CHECK_CYCLES;c++)
                        {
        *pa=0.0;
        *pr=0.0;
        for (;p<p_end;p++)
        {
        *pa+=*p;
        *pr+=SQUARE (*p);
        }
        *pa=*pa/SAMPLES_PER_CYCLE;
        *pr=*pr/SAMPLES_PER_CYCLE;
        *pr=sqrt(*pr);
                        pa++;
                        pr++;
                        p_end=p+SAMPLES_PER_CYCLE;
        }
        }

float root_mean_square (cycle)
        float cycle[];
        {
        float *p,*p_end,r;
        r=0.0;
        p=cycle;
        p_end=p+SAMPLES_PER_CYCLE;
        for (;p<p_end;++p)
        {
        r+=SQUARE(*p);
        }
        r=sqrt(r/SAMPLES_PER_CYCLE);
/*      printf("r    =%10.3f \n",r); */
                        return r;
        }


integer_order (number)
        int number;
        {
        int l=0,order=0,x=number;
        while (l<6)
        {
        x=x/10;
        if (x == 0)
        return(order);
        else
        ++order;
        }
        return order;
        }

integer_to_string (number,order,str)
        int number,order;
        char str[];
        {
        int l=order;
        while (l>=0)
        {
        str[l]=(char)(number%10+48);
        number=number/10;
```

```
—l;
}
str[order+1]='\0';
}
```

```
                    /* test                                    Aug 12 1991 */

test ()
                    {
      FILE *in_f, *out_f, *fopen (), *fclose ();
                    double  (*TF)();
                    extern double  Sigmoid();


                    int count1,count2,l,n,zoz=0,num=1,step,kiesh=0,bes=0,bev=0;
                    int cykl,y,i,points=0,order,delay=0,rem,buf,se=0,target=1;


      float *p,*p_end,*p1,*p1_end,*p2,*p2_end,*p3,*p3_end,*p4,*p4_end;
      double *pointer,*pointer_end;
                    float r1,r2,k=0.05,m=0.001,tc=0.0167;


                        double    disturb[TOTAL+TOTAL/SAMPLES_PER_CYCLE],output[CHECK_CYCLES],weights[WEI],nor[TOTAL+TOTAL/SAM-
PLES_PER_CYCLE],nn[TOTAL],result[CHECK_CYCLES],temp,ginger[ARRAY_SIZE];
                    float avg[CHECK_CYCLES],rms[CHECK_CYCLES];


/*
                    printf ("Enter patterns target : ");
                    scanf ("%d",&target);
                    printf ("\n");
*/


TF = Sigmoid;
                    j=0;
                    i=0;
                    count1=1;
                    p=array_1;
      p_end=p+Pn;
      for (;p<p_end;)
      {
back:                 if (absolute(*p)<THRESHOLD)
                    {
/*                  printf ("1.j=%d ,  p=%f , p1=%f\n",j,*p,*(p+1)); */
                    ++p;
                    ++j;
                    ++se;
                    if (se >= 32)
                    {
                    p1=store_1;
      p1_end=p1+SAMPLES_PER_CYCLE;
                    for (;p1<p1_end;)
                    {
                    *p1++=0.0;
                    }
                    rms_1=0.0;
                    r1=k;
                    r2=k/2.0;
                    goto again;
                    }
                    goto back;
                    }
/*                  printf ("2.j=%d ,  p=%f , p1=%f\n",j,*p,*(p+1)); */
      if ( ( *p == 0)  && (*(p+1) > 0.0 ) )
      goto cal_i;
      else
      if ( (*p < m) && ((*p > 0.0) && (*(p+1) > 0.0 )) )
      goto cal_i;
      else
get_v:  ++p;
                    ++j;
/*                  printf ("3.j=%d ,  p=%f , p1=%f\n",j,*p,*(p+1)); */
      if ( ( *p == 0)  && (*(p+1) > 0.0 ) )
      goto cal_i;
      else
      if ( ( ((*p)*(*(p-1)) < 0.0)  || (*p < m) ) && (*p > 0.0) )
      goto cal_i;
      else
      goto get_v;
      }


cal_i:                p1=store_1;
                    p1_end=p1+SAMPLES_PER_CYCLE;
                    p4=store_3;
                    for (;p1<p1_end;j++,p1++,p++,p4++)
                    {
                    *p1=*p;
                    *p4=*p1;
                    *(p4+SAMPLES_PER_CYCLE)=*p1;
                    }


/*
                    for(l=0;l<SAMPLES_PER_CYCLE;l++)
                    printf("s1=%7.4f , s3=%7.4f , s3s=%7.4f\n",store_1[l],store_3[l],store_3[l+SAMPLES_PER_CYCLE]);
*/


                    rms_1=root_mean_square (store_1);
/*
                    printf ("NOISE=%f    rms_1=%f\n",NOISE,rms_1);
*/
                    if (rms_1<NOISE)
                    goto get_v;
/*
                    printf ("zoz=%d , num=%d\n",zoz,num);
```

```
*/
                    ++zoz;
                    if (zoz<num)
                    goto get_v;
                    r1=(1+k)*rms_1;
                    r2=(1-k)*rms_1;
/*
                    printf ("j=%7d , rms_1=%10.3f , r1=%10.3f , r2=%10.3f\n",j,rms_1,r1,r2);
*/

again:              p2=store_2;
                    p2_end=p2+SAMPLES_PER_CYCLE;
                    for (;p2<p2_end;j++)
                    {
                    if (j  == (Pn))
                    goto end;
                    *p2++=*p++;
                    }

                    rms_2=root_mean_square (store_2);
/*
                    printf ("j=%7d , rms_2=%10.3f , count=%7d\n",j,rms_2,count1++);
*/
    if (rms_2<NOISE)
                    goto again;

                    /* add if rms_1 or rms_2 < noise_threshold => neglegt store_2 */

                    if ((rms_2<r1)&&(rms_2>r2))
    goto again;
/*
                    p=p-SAMPLES_PER_CYCLE;

                    for (l=0;l<SAMPLES_PER_CYCLE;l++)
                    printf ("j=%7d , store_1=%10.3f , store_2=%10.3f , p=%10.3f\n",j,store_1[l],store_2[l],*p++);
*/
                    p=p-SAMPLES_PER_CYCLE;
                    delay=j-SAMPLES_PER_CYCLE;
                    cykl=j/SAMPLES_PER_CYCLE;
printf("cykl=%d \n",cykl);
    for (l=0;l<cykl;l++)
                    ginger[l]=0.0;
n_set:              pointer=disturb;
                    Co=0;
                    n=0;
                    buf=delay;
                    ++l;
                    for (c=Co;c<CHECK_CYCLES;++c)
                    {
                    p1=store_1;
                    pointer_end=pointer+SAMPLES_PER_CYCLE;
                    for (;pointer<pointer_end;pointer++,p++,p1++,j++,n++,points++)
                    {
                    if (j  == Pn)
    goto end;
                    *pointer=*p-*p1;
/*
                    printf ("j=%7d , n=%7d , disturb=%10.3f , p=%10.3f , p1=%10.3f\n",j,n,*pointer,*p,*p1);
*/
                    }
                    }

                    printf ("\n I am here\n ");
                    printf ("j=%7d , delay=%7d , points=%7d , p=%10.3f , p1=%10.3f \n ",j,delay,points,*p,*p1);

there:              feature (disturb,avg,rms);

                    rows=TOTAL;columns=1;
/*
                    time=1.0;
*/
                    time=SEC_PER_SAMPLE;
                    rem=0;

/*
                    order=integer_order(l);
                    integer_to_string (l,order,str);
                    concatenate (".disturb_",str,in_name);
    concatenate (name,in_name,out_name);


    open_write_file (out_name);
    print_column_matrix_double (disturb,rows,columns,delay,rem);
    close_write_file (out_name);
    printf ("\n");
*/

    concatenate (name,".dis",out_name);
    if (i==1)
    open_write_file (out_name);
    else open_append_file (out_name);
    print_column_matrix_double (disturb,rows,columns,delay,rem);
    if (i==1)
    close_write_file (out_name);
    else close_append_file (out_name);
```

```
                    for (l=0;l<CHECK_SAMPLES;l++)
                    record_3[l]=disturb[l]+DC;

                    if (i==1 && kiesh==0)
                    {
                    get_cycle_d (record_3,array_3);
                    bes=TOTAL-jj;
                    bev=Pnn*SAMPLES_PER_CYCLE-jj;
                    printf ("\n current zero obtained  bes=%d  bev=%5d\n",bes,bev);
                    p=p-bes;
                    kiesh++;
                    --i;
                    j=j-bes;
                    points=0;
                    delay=j-SAMPLES_PER_CYCLE;

                    printf ("j=%7d , delay=%7d , points=%7d , p=%10.3f , p1=%10.3f \n ",j,delay,points,*p,*p1);

                    p4=&store_3[SAMPLES_PER_CYCLE-bev];
                    p1=store_1;
                    p1_end=p1+SAMPLES_PER_CYCLE;
                    for (;p1<p1_end;p1++,p4++)
                    {
                    *p1=*p4;
/*
                    printf("s1=%7.4f\n",*p1);
*/
                    }

                    }
                    else
                    {
    for (l=0;l<CHECK_SAMPLES;l++)
    array_3[l]=record_3[l];
                    Pnn=0;
                    }
                    if (i==0)
                    goto n_set;
                    Cn=CHECK_CYCLES;
                    printf ("i-%5d Cn=%5d Pnn=%5d\n",i,Cn,Pnn);

    normalize_d2 (array_3,nor,nn);
                    printf ("\n input normalized. \n");

/* adjust for jj */

    rem=0;
    rows=TOTAL;
                    columns=1;
/*
    time=1.0;
*/
/*
    concatenate (".nn_",str,in_name);
    concatenate (name,in_name,out_name);
    open_write_file (out_name);
    print_column_matrix_double (nn,rows,columns,buf,rem);
    close_write_file (out_name);
    printf ("\n");
*/

    concatenate (name,".nn",out_name);
    if (i==1)
    open_write_file (out_name);
    else open_append_file (out_name);
    print_column_matrix_double (nn,rows,columns,buf,rem);
    if (i==1)
    close_write_file (out_name);
    else close_append_file (out_name);

/*
    time=(float) SAMPLES_PER_CYCLE;
    rem=SAMPLES_PER_CYCLE-1;
*/

                /* Pattern recognition */

    open_read_file ("y6.wei");
    read_weights (weights);
    close_read_file ("y6.wei");

                    y=0;
    for (l=0;l<Cn;l++)
    {
    if (BP3Layer(INPUT, 6, 1, (double *) weights,(double *) &nor[y], (double *) &output[l], TF))
    {
    printf("\nError occurs in 'BP3Layer' function\n");
    exit(-1);
    }
                    y+=INPUT;
    }

    rows=Cn,columns=1;
```

```
/*
        concatenate (".res6_",str,in_name);
        concatenate (name,in_name,out_name);
        open_write_file (out_name);
        print_column_matrix_double (output,rows,columns,delay,rem);
        close_write_file (out_name);
        printf ("\n");


        concatenate (name,".res",out_name);
        if (i==1)
        open_write_file (out_name);
        else open_append_file (out_name);
        print_column_matrix_double (output,rows,columns,delay,rem);
        if (i==1)
        close_write_file (out_name);
        else close_append_file (out_name);

*/


        concatenate (name,".res",out_name);
        if (i==1)
        open_write_file (out_name);
        else open_append_file (out_name);
        print_column_matrix_double_time (output,rows,columns,delay,rem);
        if (i==1)
        close_write_file (out_name);
        else close_append_file (out_name);




        for (column=0;column<columns;++column)
        {
        step=Cn*column;
                        if (i==1)
/*
        result[step]=output[step];
*/
        result[step]=0.0;
                        else result[step]=temp;
        for (c=Co;c<Cn-1;++c)
        {
        y=step+c+1;
        result[y]=result[y-1]+tc*(output[y]-result[y-1]);
        if (result[y] > 1.0)
        result[y]=1.0;
        }
                        temp=result[y];
/*
                        printf ("\ny=%5d , output=%lf , result=%lf , temp=%lf\n",y,output[y],result[y],temp);
*/
        }


/*
                        concatenate (name,".int",out_name);
                        if (i==1)
                        {
        open_write_file (out_name);
                        for (l=0;l<j;l++)
                        fprintf (out_f,"%7.3e \t 0.0\n",l*TIME_C);
        print_column_matrix_double (result,rows,columns,delay,rem);
                        }
                        else open_append_file (out_name);
        print_column_matrix_double (result,rows,columns,delay,rem);
        if (i==1)
        close_write_file (out_name);
                        else close_append_file (out_name);
*/


        concatenate (name,".int",out_name);
        if (i==1)
                        {
        open_write_file (out_name);
                        print_column_matrix_double_time (ginger,cykl,1,0,0);
/*
*/
                        }
        else
                        open_append_file (out_name);
        print_column_matrix_double_time (result,rows,columns,delay,rem);
        if (i==1)
        close_write_file (out_name);
        else close_append_file (out_name);


                        if (i<LAST_SET)
                        {
                        j=buf+CHECK_SAMPLES;
                        delay=j;
                        goto n_set;
                        }
                        else
                        {
                        printf ("Last set\n");
                        return;
                        }
end:                    printf ("End of file.\n");
```

```
                }

normalize_d (array,record)
        double array[],record[];
        {
        double *p1,*p1_end,*p2,max;
        p1=array;
        p2=record;
        Co=0;

        for (c=Co;c<Cn;c++)
        {
        p1_end=p1+SAMPLES_PER_CYCLE;
        max=0.0;
        for (j=0;p1<p1_end;p1++,j++)
        {
        a=*p1;
        b=absolute(a);
        if (b>max)
        max=b;
        }
        p1=p1_end-SAMPLES_PER_CYCLE;
        for (j=0;p1<p1_end;p1++,p2++,j++)
        {
        *p2=(*p1+max)/(2.0*max);
/*
                printf ("j=%5d , p1=%10.5f , p2=%10.5f\n",j,*p1,*p2);
*/
        }
        }
        }


normalize_d2 (array,record,nn)
        double array[],record[],nn[];
        {
        double *p1,*p1_end,*p2,*p3,max;
        p1=array;
        p2=record;
        p3=nn;
                Co=0;

        for (c=Co;c<Cn;c++)
        {
        p1_end=p1+SAMPLES_PER_CYCLE;
        max=0.0;
        for (j=0;p1<p1_end;p1++,j++)
        {
        a=*p1;
        b=absolute(a);
        if (b>max)
        max=b;
        }

        p1=p1_end-SAMPLES_PER_CYCLE;
        for (j=0;p1<p1_end;p1++,j++)
        {
        a=*p1;
        if (absolute(a)>=LEVEL*max)
        ++width[c];
        }
        if (width[c]<=8)
        width[c]=1;
        else width[c]=0;


        p1=p1_end-SAMPLES_PER_CYCLE;
        for (j=0;p1<p1_end;p1++,p2++,p3++,j++)
        {
        *p2=(*p1+max)/(2.0*max);
                *p3=*p2;
/*              printf ("j=%5d , p1=%10.5f , p2=%10.5f ,  c=%5d , width=%5d\n",j,*p1,*p2,c,width[c]);
*/
        }
                *p2++=((double)width[c]);
        }


        }


get_cycle_d (array,record)
        double array[],record[];
        {
                double *p1,*p1_end,*p2,m=0.01,n=0.00001;
        jj=2*SAMPLES_PER_CYCLE;
        p1=&array[jj];
        p1_end=p1+CHECK_SAMPLES;
        p2=record;
        for (;p1<p1_end;)
        {
        if ( ( *p1 == 0)  && (*(p1+1) > 0.0 ) )
        goto cal_j;
        else
        if ( (*p1 < m) && ((*p1 > 0.0) && (*(p1+1) > 0.0 )) )
        goto cal_j;
        else
        if ( (absolute(*p1) < m) && (jj >= SAMPLES_PER_CYCLE)  )
        goto cal_j;
```

```
        else
get_v:  ++p1;
                        ++jj;
        if ( ( *p1 == 0)  && (*(p1+1) > 0.0 ) )
        goto cal_i;
        else
        if ( ( ((*p1)*(*(p1-1)) < 0.0)  || (*p1 < m) ) && (*p1 > 0.0) )
        goto cal_i;
        else
        goto get_v;
        }


cal_i:                  if (Pnn*SAMPLES_PER_CYCLE == jj)
                        goto cal_i2;
                        else
                        {
inc_jj:                 ++p1;++jj;
                        printf ("jj<Pnn*samples...p1=%lf  m=%f \n",*p1,m);
        if ( (*p1 < m) )
                        {
                        printf ("p1<m");
                        if (Pnn*SAMPLES_PER_CYCLE == jj)
        goto cal_i2;
                        else
                        printf ("still < Pnn*....\n");
                        goto inc_jj;
                        }
                        else
                        {
                        printf ("sorry p1>m return back. jj=%5d\n",jj);
                        --p1;--jj;
                        }
                        }


cal_i2:  for (;p1<p1_end;p1++,p2++)
        {
        *p2=*p1;
        }
        Pnn=top(jj,SAMPLES_PER_CYCLE);
        printf ("\n jj=%5d  Pnn=%5d \n",jj,Pnn);
        }
```

```
                    /* options                                  Apr 26 91 */

random ()
                    {
                    rows=Pn;columns=1;
        Cn=rows/SAMPLES_PER_CYCLE;

        dc_ac (array_1,dc,ac,ra);

        difference (dc,dc_dif);
        difference (ac,ac_dif);

        random_detect (dc_dif,dc_rnd);
        random_detect (ac_dif,ac_rnd);

        concatenate (name,dc_rndf,out_name);
        open_write_file (out_name);
        print_column_matrix (dc_rnd,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

        concatenate (name,ac_rndf,out_name);
        open_write_file (out_name);
        print_column_matrix (ac_rnd,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

                    }

all ()
                    {

                    rows=Pn;columns=1;
        Cn=rows/SAMPLES_PER_CYCLE;

        dc_ac (array_1,dc,ac,ra);

        difference (dc,dc_dif);
        difference (ac,ac_dif);

        random_detect (dc_dif,dc_rnd);
        random_detect (ac_dif,ac_rnd);

        concatenate (name,dcf,out_name);
        open_write_file (out_name);
        print_column_matrix (dc,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

        concatenate (name,acf,out_name);
        open_write_file (out_name);
        print_column_matrix (ac,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

        concatenate (name,dc_rndf,out_name);
        open_write_file (out_name);
        print_column_matrix (dc_rnd,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

        concatenate (name,ac_rndf,out_name);
        open_write_file (out_name);
        print_column_matrix (ac_rnd,Cn,columns);
        close_write_file (out_name);
        printf ("\n");

                    }
```

```
/* This is the header file  for the BackPropagation ANNs function.

   This function simulates the one hidden layer BackPropagation algorithm.
   Only forward calculation is implemented.


   The function needs number of input neurons, number of hidden neurons,
   number of output neurons, a pointer to the weight data record, a pointer
   to the input vector data record, a pointer to the output vector data
   record, and a pointer to a transfer function, as the input arguments,
   and will return an error value.

   It will return (-1) if there is an internal error, or (0) if everything
   is okay.


   Programmed by : Adi Indrayanto
   First version : 11/29/1990
   Current vers. : 2.0
   Last modific. : -

*/

int             BP3Layer(long nInput, long nHidden, long nOutput, double *weights,
                                                    double *inputVector, double *outputVector,
                                                                        double (*transFunc)(double data));

double          Sigmoid(double data);

/*  This is the BPThreeLayer function code

    Programmed by : Adi Indrayanto
    First version : 11/29/1990
    Current vers. : 2.0
    Last modific. : -

*/

#include "BP3Layer.h"

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

int  BP3Layer(long nInput, long nHidden, long nOutput, double *weights,
                                                    double *inputVector, double *outputVector,
                                                                        double (*transFunc)(double data))
{
  double  *hiddenNeurons;
  long    ix, iy;
  long    offBias1, offBias2, offBias3, offWeight2;

  if ((hiddenNeurons = (double*) calloc(nHidden, sizeof(double))) == NULL)
    return(-1);

  offWeight2 = nInput * nHidden;
  offBias1 = offWeight2 + (nHidden * nOutput);
  offBias2 = offBias1 + nInput;
  offBias3 = offBias2 + nHidden;

  for (ix = 0; ix < nHidden; ix++)
  {
                  hiddenNeurons[ix] = weights[offBias2 + ix];
                  for (iy = 0; iy < nInput; iy++)
      {
                      hiddenNeurons[ix] += weights[ix*nInput+iy] * inputVector[iy];
                  }

                  if (transFunc != NULL)
        hiddenNeurons[ix] = (*transFunc)(hiddenNeurons[ix]);
  }

 for (ix = 0; ix < nOutput; ix++)
 {
                  outputVector[ix] = weights[offBias3 + ix];
                  for (iy = 0; iy < nHidden; iy++)
                  {
                     outputVector[ix] += weights[offWeight2 + ix*nHidden + iy] *
                   hiddenNeurons[iy];
                  }

                  if (transFunc != NULL)
                    outputVector[ix] = (*transFunc)(outputVector[ix]);
  }

 free(hiddenNeurons);
 return(0);

}
```

```
/*  The Sigmoid transfer function used for BPThreeLayer function.
    Passed this function as the argument of the BPThreeLayer function.

*/

double              Sigmoid(double data)
{
  register double   result;

  result = 1/(1 + exp((-1) * data));
  return(result);
}
```

```
/* file_manipuation                    Aug 7 91 */

concatenate (str1,str2,to)
        char *str1,*str2,*to;
        {
        while (*str1)
        *to++=*str1++;
        while (*str2)
        *to++=*str2++;
        *to='\0';
        }


copy_string (from,to)
        char *from,*to;
        {
        while (*from)
        *to++=*from++;
        *to='\0';
        }

open_read_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        in_f=fopen (filename,"r");
        if (in_f == NULL)
        printf ("Can't open %s file\n",filename);
        else
        printf ("File %s is open\n",filename);
        }


close_read_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        fclose (in_f);
        printf ("File %s is closed\n",filename);
        }


open_write_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        out_f=fopen (filename,"w");
        if (out_f == NULL)
        printf ("Can't open %s file\n",filename);
        else
        printf ("File %s is open\n",filename);
        }

close_write_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        fclose (out_f);
        printf ("File %s is closed\n",filename);
        }


open_append_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        out_f=fopen (filename,"a");
        if (out_f == NULL)
        printf ("Can't open %s file\n",filename);
        else
        printf ("File %s is open\n",filename);
        }


close_append_file (filename)
        char filename[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        fclose (out_f);
        printf ("File %s is closed\n",filename);
        }
```

```
                    /* file_read                                    July 8 91 */

read_hex ()
                    {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,hex,in_name);
        open_read_file (in_name);
        hex_decimal (quantized_data,array_1);
        close_read_file (in_name);
        printf ("\n");
                    }

read_dat ()
                    {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,"",in_name);
        open_read_file (in_name);
        read_input (array_1);
        close_read_file (in_name);
        printf ("\n");
                    }

read_lotus ()
                    {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,prn,in_name);
        open_read_file (in_name);
        read_input_lotus (quantized_data);
        close_read_file (in_name);
        printf ("\n");
        }

read_lotus_2 ()
        {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,dat,in_name);
        open_read_file (in_name);
        read_input_lotus_2 (record_1);
        close_read_file (in_name);
        printf ("\n");
        }

read_lotus_out ()
        {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,".out",in_name);
        open_read_file (in_name);
        read_input_lotus_2 (array_1);
        close_read_file (in_name);
        printf ("\n");
        }


read_oct ()
        {
        printf ("ENTER FILE NAME: ");
        scanf  ("%24s", name);
        printf ("\n");
        concatenate (name,oct,in_name);
        open_read_file (in_name);
        oct_decimal (quantized_data,array_1);
        close_read_file (in_name);
        printf ("\n");
        }

read_input (array)
        float array[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        float *pointer;
        int a;
        j=0;
        pointer=array;
        do
        {
        fscanf (in_f,"%f",pointer);
        ++pointer;
        ++j;
        } while ((car=getc(in_f)) != EOF);
        Po=0;
        Pn=j-1;
        last_record=(Pn/BLOCK_SIZE);
                    printf ("Pn=%5d , last_record=%5d\n",Pn,last_record);
        }

read_input_lotus (array)
        int array[];
        {
```

```
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int *pointer;
        j=0;
        pointer=array;
        do
        {
        fscanf (in_f,"%d",pointer);
        ++pointer;
        ++j;
        } while ((car=getc(in_f)) != EOF);
        Po=0;
        Pn=j-1;
        last_record=(Pn/BLOCK_SIZE);
                        printf ("Pn=%5d , last_record=%5d\n",Pn,last_record);
        }

read_input_lotus_2 (array)
        float array[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        float *pointer;
        j=0;
        pointer=array;
        do
        {
/*
        fscanf (in_f,"%*d %f",pointer);
*/
        fscanf (in_f,"%*e %f",pointer);
        ++pointer;
        ++j;
        } while ((car=getc(in_f)) != EOF);
        Po=0;
        Pn=j-1;
        last_record=(Pn/BLOCK_SIZE);
        printf ("Pn=%5d , last_record=%5d\n",Pn,last_record);
        }


read_weights (array)
        double array[];
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        double *pointer;
        int q=0;
        pointer=array;
        do
        {
        fscanf (in_f,"%lf",pointer);
        ++pointer;
        ++q;
        } while ((car=getc(in_f)) != EOF);
        printf ("q=%5d\n",q-1);
        }
```

```
                    /* file_write                                   Aug 12 91 */

write_data ()
                    {
        concatenate (name,dat,out_name);
                    rows=Pn;columns=1;
        open_write_file (out_name);
        print_column_matrix (array_1,rows,columns);
        close_write_file (out_name);
        printf ("\n");
                    }

write_data_2 ()
        {
        concatenate (name,".x",out_name);
        rows=Pn;columns=1;
        open_write_file (out_name);
        print_column_matrix (array_1,rows,columns);
        close_write_file (out_name);
        printf ("\n");
        }

write_mbplot ()
        {
        concatenate (name,".mb",out_name);
        rows=Pn;columns=1;
                    Cn=Pn/SAMPLES_PER_CYCLE;Co=0;Po=0;
        open_write_file (out_name);
        print_mbplot (array_1,rows,columns);
        close_write_file (out_name);
        printf ("\n");
        }


write_arrange ()
                    {
        concatenate (name,rec,out_name);
        open_write_file (out_name);
                    for (j=0;j<Pn;++j)
                    fprintf (out_f,"%7.3e %10.3f\n",j*time,array_1[j]);
        close_write_file (out_name);
        printf ("\n");
                    }



print_column_matrix_double_3 (array,rows,columns,de,rem)
        double array[];
        int rows,columns,de,rem;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column,d;
        double *p,*p_end;
        p=array;
        for (row=0;row<rows;++row)
        {
        fprintf (out_f,"%7.3e  ",de*time+row*SAMPLES_PER_CYCLE+rem);
        for (column=0;column<columns;++column)
        {
        d=column*rows+row;
        p=&array[d];
        fprintf (out_f,"%10.3lg",*p);
        }
        fprintf (out_f,"\n");
        }
        }


print_column_matrix_double (array,rows,columns,de,rem)
        double array[];
        int rows,columns,de,rem;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column,d;
        double *p,*p_end;
        p=array;
        for (row=0;row<rows;++row)
        {
        fprintf (out_f,"%7.3e  ",(de+row)*time+rem);
        for (column=0;column<columns;++column)
        {
        d=column*rows+row;
        p=&array[d];
        fprintf (out_f,"%10.3lg",*p);
        }
        fprintf (out_f,"\n");
        }
        }


print_column_matrix_double_time (array,rows,columns,de,rem)
        double array[];
        int rows,columns,de,rem;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column,d;
        double *p,*p_end;
        p=array;
```

```
        for (row=0;row<rows;++row)
        {
        fprintf (out_f,"%7.3e  ",de*TIME_S+(row+1)*TIME_C);
        for (column=0;column<columns;++column)
        {
        d=column*rows+row;
        p=&array[d];
        fprintf (out_f,"%10.3lg",*p);
        }
        fprintf (out_f,"\n");
        }
        }


print_column_matrix_dis (array,rows,columns,de,rem)
                float array[];
        int rows,columns,de,rem;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
        for (row=0;row<rows;++row)
        {
        fprintf (out_f,"%7.3e",de+row*time+rem);
        for (column=0;column<columns;++column)
        {
        j=column*rows+row;
        p=&array[j];
        fprintf (out_f,"%10.3f",*p);
        }
        fprintf (out_f,"\n");
        }
        }

print_column_matrix (array,rows,columns)
        float array[];
        int rows,columns;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
        for (row=0;row<rows;++row)
        {
        fprintf (out_f,"%7.3e",row*time);
        for (column=0;column<columns;++column)
        {
        j=column*rows+row;
        p=&array[j];
        fprintf (out_f,"%10.3f",*p);
        }
        fprintf (out_f,"\n");
        }
        }

print_row_matrix (array,rows,columns)
        float array[];
        int rows,columns;
        {

        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
        for (row=0;row<columns;++row)
        {
        fprintf (out_f,"%7d",row);
        for (column=0;column<rows;++column)
        {
        fprintf (out_f,"%10.3f",*p);
        *p++;
        }
        fprintf (out_f," 1\n");
        }
        }


print_row_matrix_nn (array,rows,columns)
        float array[];
        int rows,columns;
        {

        /* Prints in NN Training Format */

        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
        for (row=0;row<columns;++row)
        {
        fprintf (out_f,"%s_%02d ",in_name,row);
        for (column=0;column<rows;++column)
        {
        fprintf (out_f,"%6.4f ",*p);
        *p++;
        }
        fprintf (out_f," %d\n",target);
        }
        }
```

```
print_row_matrix_nn2 (array,width,rows,columns)
        float array[];
        int rows,columns,width[];
        {

        /* Prints in NN Training Format */

        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
        for (row=0;row<columns;++row)
        {
        fprintf (out_f,"%s_%02d ",in_name,row);
        for (column=0;column<rows;++column)
        {
        fprintf (out_f,"%6.4f ",*p);
        *p++;
        }
        fprintf (out_f," %d %d\n",width[row],target);
        }
        }


print_row_matrix_nn3 (array,target,rows,columns)
        double array[];
        int rows,columns,target;
        {

        /* Prints in NN Training Format */

        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        double *p,*p_end;
        p=array;
        for (row=0;row<columns;++row)
        {
        fprintf (out_f,"%s_%02d ",in_name,row);
        for (column=0;column<rows;++column)
        {
        fprintf (out_f,"%6.4g ",*p);
        *p++;
        }
        fprintf (out_f," %d\n",target);
        }
        }

print_mbplot (array,rows,columns)
                float array[];
        int rows,columns;
                {


        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column;
        float *p,*p_end;
        p=array;
                fprintf (out_f,"%s\n",in_name);
                fprintf (out_f,"%7d %7d %7d CYCLES\n",Cn,Co,1);
                fprintf (out_f,"%7d %7d %7d SAMPLES\n",SAMPLES_PER_CYCLE,Po,1);
                fprintf (out_f,"a b c\n");

                for (c=Co;c<Cn;c++)
                {
        for (row=0;row<SAMPLES_PER_CYCLE;++row)
        {
        fprintf (out_f,"%7d %02d %10.3f\n",c,row,*p);
        *p++;
        }
        }
        }


rearrange_array (array,record,rows,columns,step)
        float array[];
        float record[];
        int rows,columns,step;
        {
        FILE *in_f, *out_f, *fopen (), *fclose ();
        int row,column,junk,y,t;
        float *p,*p_end;
        float *pointer, *pointer_end;
        j=0;
        p=record;
        junk=Pn-columns*rows;
                y=0;
                t=0;
        for (column=0;column<columns;++column)
        {
                pointer=&array[y];
        pointer_end=pointer+rows;
        for (;pointer<pointer_end;)
        {
                if (j+junk <= Pn)
        {
        *p++=*pointer++;
        j++;
        }
                else
                break;
```

```
}
        y=y+rows+JUNK-t++;
}
}
```

B Security Testing Algorithm: Program List

```
=============================================
= THE TESTING OF MICROPROCESSOR ALGORITHMS FOR =
=   HIGH IMPEDANCE FAULT/LOAD DISCRIMINATION   =
=============================================
      Using MathCAD Software and Notation
      ***********************************

MathCAD FILE : mcad-prog

IV := READPRN(data)                    read current & voltage samples

n := rows(IV)        n =               total number of samples

                     s := 32           samples per cycle

                     i := 0 ..s - 1    counter of sample number in a cycle

c := floor(n/s)      c =               number of cycles

                     j := 1 ..c        counter of cycle number

I := IV<0>           V := IV<1>        current and voltage vectors


                          ==================
                          = DATA PROCESSING =
                          ==================

FFT OF THE CURRENT SIGNAL
=========================

k := 2/sqrt(s)                         factor to get actual amplitude of the harmonics

A     := I
 j,i     i+32·(j-1)                    matrix of cycles(row) x  samples(column)

B := A^T                               transpose for parallel processing

Idft<j>  := |fft(B<j>)|·k              take FFT for each cycle

x := last(Idft<1>)                     index of last element in the FFT

y := 0 ..x                             counter of harmonics per cycle

FFT OF THE VOLTAGE SIGNAL
=========================

C     := V
 j,i     i+32·(j-1)

D := C^T

Vdft<j>  := |fft(D<j>)|·k
```

```
============================
= THIRD HARMONIC ALGORITHM =
============================

THIRD HARMONIC CURRENT MAGNITUDE AND PHASE PER CYCLE
===================================================

mI3_j := |Idft_3^<j>|                                   third harmonic current magnitude

φ_j := if[Idft_3^<j> ≈ 0,0,arg[Idft_3^<j>]]

aI3_j := arg[Vdft_1^<j>] - φ_j                          third harmonic current phase

I3_j := mI3_j ·cos[aI3_j] + i ·mI3_j ·sin[aI3_j]        third harmonic current vector

                                                        angles are in harmonics rads.
                                                        magnitudes are in per unit

SIGNAL AVERAGING
================

Short time average
------------------
Ks := 0.1                                               time constant

As_1 := 0.9·I3_1                                        initial value

As_j := if[j > 1,As_(j-1)·Ks + I3_j·(1 - Ks),As_1]      current averaging

Long time average
-----------------
Kl := 0.9                                               time constant

Al_1 := 0.1·I3_1                                        initial value

Al_j := if[j > 1,Al_(j-1)·Kl + I3_j·(1 - Kl),Al_1]      current averaging

PHASOR CHANGE
=============
DI3_1 := As_1                                           initial value

DI3_j := if[j > 1,As_j - Al_(j-1),DI3_1]                change in magnitude

aDI3_j := if[DI3_j ≈ 0,0,arg[DI3_j]]                    change in phase
```

```
==========================
=  POWER RATIO ALGORITHM =
==========================
```

$sdft := \overline{(|Idft| \cdot |Idft|)}$ individual harmonic power

$e := 6,8 \ldots x - 2$ counter of even harmonics

$o := 7,9 \ldots x - 1$ counter of odd harmonics

$ee_j := \sum_e sdft_e^{\langle j \rangle} \cdot 10^4$ sum of even harmonics power per cycle

$oe_j := \sum_o sdft_o^{\langle j \rangle} \cdot 10^4$ sum of odd harmonics power per cycle

even-to-odd harmonics power ratio per cycle is

$$r_j := if\left[ee_j < 1.0, 0, if\left[oe_j < 1.0, 0, \frac{ee_j}{oe_j}\right]\right]$$

```
====================
= RESULTS STORAGE =
====================
```

$R_{j,0} := j$ cycle number

$R_{j,1} := |DI3_j|$ change in third harmonic current magnitude (pu)

$R_{j,2} := if\left[|DI3_j| < 0.001, 0, aDI3_j \cdot \frac{180}{\pi} + 6.77\right]$ change in third harmonic current phase (degrees)

$R_{j,3} := r_j$ even-to-odd harmonics power ratio

WRITEPRN(results) := R write results to disc

Biography

Ahmad F. Sultan (IEEE Student Member '82) was born in Alexandria, Egypt, in 1961. He received the B.Sc. (First class honors, top of the class) and M.Sc. degrees in Electrical Engineering from Ain–Shams University, Cairo–Egypt, in '83 and '87 respectively. He worked in ASEA for three years, and was a teaching assistant, then an assistant lecturer at Ain–Shams University ('83 to '88). He worked for his Ph.D. degree at the University of Manitoba since January '89. He was the chief demonstrator of the Electrical Machines laboratory at the University of Manitoba during his graduate study.

Mr. Sultan has been a recipient of a number of graduate and under graduate awards and fellowships: the University of Manitoba Graduate Fellowship ('90 and '91); the D.M. Stephens Memorial Fellowship '90; Ain–Shams University M.Sc. Award '87; and Ain–Shams University Undergraduate Fellowship ('78 to '83).

His career interests are in neural networks, computer vision, robotics, optical systems, power systems, and education. Social activities include travelling and horseback riding with the family; sport parachuting, was an assistant instructor in Egypt; athletics and sea.