Dielectrometry Measurements of Moisture Dynamics in Oil-Impregnated Pressboard

by

Yanko Konstantinov Sheiretov

B.S., Massachusetts Institute of Technology (1992)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in Electrical Engineering

and

Electrical Engineer

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .....
Department of Electrical Engineering and Computer Science
May 11, 1994

Certified by .....
Markus Zahn
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Dielectrometry Measurements of Moisture Dynamics in Oil-Impregnated Pressboard

by

Yanko Konstantinov Sheiretov

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Science in Electrical Engineering
and
Electrical Engineer

Abstract

The dielectric spectrum of pressboard is a function of its moisture content and temperature. The real component of the complex permittivity gives the dielectric constant while the imaginary component characterizes the power dissipation in the material. In oil-impregnated pressboard of medium and low humidity the dielectric spectrum's shape and amplitude do not change with variations in temperature and moisture content, but only shift in frequency. Thus it is possible to create a universal curve, with appropriate temperature correction factors, which can be used to extract information about the moisture dynamics of solid transformer insulation from dielectrometry measurements.

Such measurements may be taken with the material placed in a parallel-plate lossy capacitor structure whose complex impedance is measured. In this way one obtains values for the complex permittivity of the material that are averaged across its thickness. An alternative technique, known as imposed ω-k dielectrometry, uses a set of interdigitated electrodes on one surface of the material. The electric field has only a limited depth of penetration, which is determined by the spacing of the electrodes. Therefore, if measurements are taken at more than one spatial wavelength, one obtains information about the one-dimensional spatial profile of the complex permittivity.

Measurements using the parallel-plate methodology establish a mapping of the dielectric spectrum of EHV-Weidmann HIVAL pressboard impregnated with Shell Diala A transformer oil, as a function of temperature and water content. This mapping is then used to determine spatial moisture profiles in pressboard in other experiments which make use of a three-wavelength interdigitated sensor.

Thesis Supervisor: Markus Zahn
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

The research presented in this thesis was carried out at the Laboratory of Electromagnetic and Electronic Systems at the Massachusetts Institute of Technology. It was supported by the Electric Power Research Institute (RP-1289-5) under the management of Mr. S. R. Lindgren. The thesis was supervised by Professor Markus Zahn at the Massachusetts Institute of Technology.

I would like to thank Prof. Zahn for everything that I have learned from him over the past two years. In addition to giving me direct guidance with my work, without which this thesis would not have been possible, Prof. Zahn taught me to strive for perfection in everything I do. I would also like to thank him for all the advice and support I have received from him, and for always finding time to talk to me.

Dr. Philip von Guggenberg has also helped me immeasurably with my research. I have had the opportunity to take advantage of his vast knowledge in the field of dielectrometry and to be able to discuss with him any problems with my research. Many times he has volunteered to review my work and I always found his input of great value. Much of the research presented in this thesis is based on previous work done at MIT by Dr. M. Zaretsky and Dr. P. A. von Guggenberg. I was glad that I could discuss some of my work directly with both.

Up to this day, whenever I have a question about any aspect of my work — be it theoretical, mathematical, computer-related, or mechanical — I go to my lab partner Andrew Washabaugh, who always manages to find the answer or direct me to a resource. I thank him for his willingness to take time to discuss problems with me. On several occasions he has dedicated hours of his time to work on theoretical problems with which I needed help.

I would also like to thank the entire LEES staff, and in particular Mr. Paul Warren, Ms. Kathy McCue, and Mr. Wayne Ryan, for their support with technical, administrative, and personal issues.

Finally, I would like to thank my friends for helping me make it through a difficult year and for being patient with me during these last several very busy months.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In this thesis we discuss dielectrometry measurements of insulating materials, with an emphasis on solid and liquid transformer insulation, and their application to the measurement of the moisture content of these materials.

Monitoring the condition of the insulation is of particular importance to high-power transformers, where the insulating materials are subjected to high levels of electrical and thermal stress.

1.1.1 High Power Transformers

High-power transformers are an essential element in the distribution of electrical energy. The demand for energy is perpetually increasing, placing ever tougher requirements on the performance characteristics of these transformers. The transmission of greater quantities of electrical energy affects the operation of the transformers by requiring efficient transmission of more energy at higher voltages, which in turn subjects transformer insulation to higher levels of electrical stress. In addition, the heat dissipation due to losses in the transformer cores and windings requires higher coolant speeds, which in turn increases the level of static electrification.

In the last decade it has become desirable to be able to monitor closely the condi-

tion of high-power transformers, because they have been pushed to their limits, which is reflected in the increase of transformer failures. The need for greater efficiency has reduced the margin of safety in the operation of the transformers, making it very important to identify and predict critical conditions that may lead to failures.

The presence of moisture in the solid and liquid transformer insulation, i.e. pressboa d and oil, is a major factor that affects the operation of the transformers. Although moisture does not seem to greatly affect the conductivity of the oil, it reduces its dielectric strength. Moisture also affects the conductivity of the pressboard, which in turn increases the dissipated power and the rate of static charge relaxation, which is a crucial factor in static electrification phenomena.

Load transients which transformers undergo, especially upon power-up, cause rapid changes in the insulation's temperature. Temperature affects the solubility equilibrium of moisture between the solid and liquid insulation and also directly influences the insulation's conductivity. Moisture in the oil may under decreasing temperature transients result in free water that can lead to electrical breakdown. A mass transfer process of water results from the equilibrium imbalance, in which at higher temperatures moisture leaves the pressboard to enter the oil. The oil establishes moisture equilibrium with an interfacial zone at the surface of the pressboard. The steady state is reached when moisture from deep inside the pressboard diffuses to the surface to establish a uniform moisture distribution. The transient interfacial dry zones are highly insulating, and as a consequence significant surface charge can accumulate to cause surface spark discharges. Such critical conditions can lead to a high level of static electrification and possibly catastrophic failure of the unit. It is therefore important to be able to monitor the moisture dynamics in such systems, in order to understand the failure mechanisms and to prevent critical conditions.

1.1.2 Other Applications

The dielectrometry methods developed specifically for pressboard have applications in many other fields also: The dielectric properties of a material are greatly affected by many of its other physical properties, such as temperature, pressure, mechanical

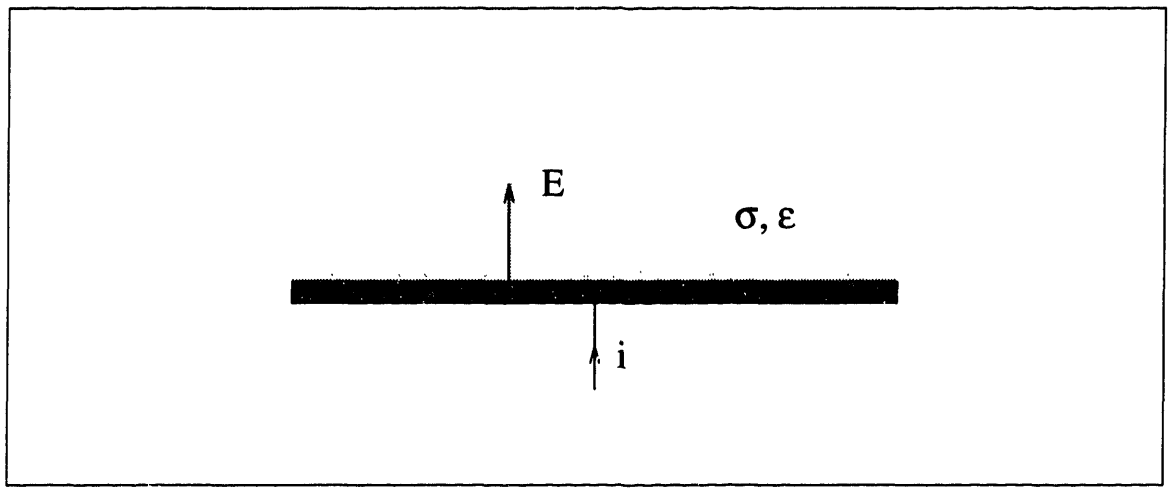

Figure 1-1: Terminal current of an electrode in contact with a conducting dielectric medium

stress, etc. In polymers, the dielectric constant may be related to the degree of polymerization. There are many applications in quality control, where deviations in the dielectric properties of a material may correspond to flaws in its structure.

As materials age, their dielectric constant and conductivity may change too. In general, whenever the condition of a dielectric material must be monitored, dielectrometry measurements provide a simple, non-destructive, real-time measurement, which can be related to the property in question.

1.2 Dielectric Properties of Materials

There are two parameters of a medium that determine the quasi-static distribution of electric fields: the dielectric permittivity ϵ and the conductivity σ. The former determines the displacement current density from the electric field, while the latter relates the conduction current density to the electric field. The permittivity governs energy storage (reactive power) phenomena, while the conductivity determines the power dissipation (active power).

Consider an electrode in contact with a medium as shown in Figure 1-1. Since the total current density due to conduction and displacement is in the same direction as the electric field, we will drop the vector signs in the following discussion. In the

one-dimensional geometry of Figure 1-1, the current densities and the electric field are perpendicular to the electrode. Let the electric field at the electrode surface be E. We are interested in the total terminal current per unit electrode area J that flows into the electrode. Integrated over the electrode area, this would yield the terminal current:

$$i = \int_S J da \tag{1.1}$$

The component of J due to conduction follows the ohmic constitutive law:

$$J_c = \sigma E \tag{1.2}$$

The displacement current density arises from the buildup of surface charge σ_s at the electrode:

$$J_d = \frac{d\sigma_s}{dt} = \frac{d}{dt}(\epsilon E) \tag{1.3}$$

The total terminal current per unit electrode area is then:

$$J = J_c + J_d = \sigma E + \frac{d}{dt}(\epsilon E) \tag{1.4}$$

If the system is under AC steady-state operation, every quantity $F(t)$ may be expressed as:

$$F(x,y,z,t) = \Re\left\{\widehat{F}(x,y,z)e^{j\omega t}\right\} \tag{1.5}$$

where ω is the radian frequency of excitation. If ϵ is constant with time, we may rewrite equation 1.3 in terms of complex amplitudes as:

$$\widehat{J}_d = j\omega\epsilon\widehat{E} \tag{1.6}$$

For the total current density we may then write:

$$\widehat{J} = \widehat{J}_d + \widehat{J}_c = j\omega\epsilon\widehat{E} + \sigma\widehat{E} = j\omega\widehat{E}\left(\epsilon + \frac{\sigma}{j\omega}\right) \tag{1.7}$$

It is convenient to define the *complex permittivity* ϵ^* of a medium as:

$$\epsilon^* = \epsilon' - j\epsilon'' = \epsilon - j\frac{\sigma}{\omega} \tag{1.8}$$

which lets us rewrite equation 1.7 in a form similar to equation 1.6, thus combining conduction phenomena with polarization phenomena:

$$\widehat{J} = j\omega\epsilon^*\widehat{E} \tag{1.9}$$

We shall use this definition of the complex permittivity throughout this thesis.

1.2.1 Dielectric Spectra

The *dielectric spectrum* of a material is a representation of its complex permittivity, $\epsilon^* = \epsilon' - j\epsilon''$, as a function of frequency. The real component ϵ' gives the dielectric constant while the imaginary component ϵ'' determines the power dissipation (loss) in the material.

Once it is known how the dielectric spectrum of oil-impregnated pressboard varies with temperature and moisture, it is possible to measure the moisture content in a sample by taking a frequency scan and comparing the results to the known calibration mapping. This type of mapping is unique to every type of paper and may depend on the amount of impurities in it. Such a mapping for pressboard is presented in Chapter 2.

1.2.2 Kramers-Krönig Relations

The Kramers-Krönig relations link the real and imaginary components of the dispersive part of the complex permittivity, defined in equation 1.8. As a direct consequence of causality, the following equations hold:

$$\chi'(\omega) = \frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty}\frac{\chi''(x)}{x-\omega}dx \tag{1.10}$$

$$\chi''(\omega) = -\frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty}\frac{\chi'(x)}{x-\omega}dx \tag{1.11}$$

where the real and imaginary parts of the dispersive part of the dielectric susceptibility χ^* are defined as follows:

$$\epsilon' = \epsilon = \epsilon_0\chi' + \epsilon_\infty \tag{1.12}$$

$$\epsilon'' = \frac{\sigma}{\omega} = \epsilon_0\chi'' + \frac{\sigma_0}{\omega} \tag{1.13}$$

$$\chi^* = \chi' - j\chi'' \tag{1.14}$$

Appendix A presents the derivation of these relations and some of their consequences. In this section we discuss what the Kramers-Krönig relations can tell us about the dielectric spectra of materials.

In an *ohmic* material, ϵ and σ are independent of the frequency or amplitude of the applied electric field and a plot of $\log(\epsilon''/\epsilon_0)$ versus $\log\omega$ has a slope of -1. As discussed in Appendix A, for such materials $\chi^* = 0$.

In a *dispersive* material, when ϵ'' is plotted against frequency on a log-log scale, it can be characterized by one or more *loss peaks.* The magnitude of the slope at which these peaks are approached on either side is between 0 and 1 for most materials [1, pp. 163–200]. For every loss peak in the ϵ'' spectrum, there is an associated elevation in the ϵ' spectrum proportional to the area under the corresponding peak in ϵ'' [1, pp. 47–52]. This is illustrated in Figure 1-2.

1.3 Moisture Dynamic Processes in Pressboard/Oil Systems

Section 1.1.1 discussed the significance of the presence of moisture in solid and liquid transformer insulation. During thermal transients complex dynamic processes occur as temperature gradients develop. Temperature transients disturb the moisture

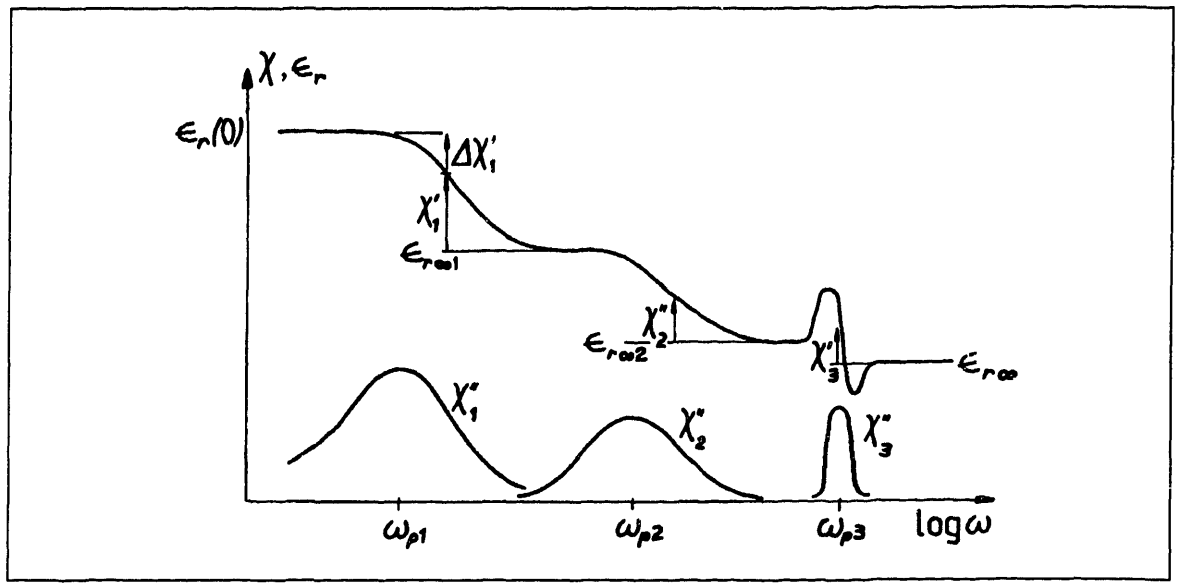

Figure 1-2: An illustration of the manner in which the real part of the complex permittivity is made up of contributions from all loss processes [1, pp. 50]

equilibrium of the system, causing the initiation of moisture mass transfer processes. Transformer oil and pressboard are very dissimilar materials, in that the former is hydrophobic and the latter is hydrophilic. Typical values for the water content of pressboard are 0.5–5%, while in oil at room temperature the saturation moisture content is about 50 ppm (parts per million). As a consequence, almost all of the moisture present in the system resides in the pressboard. As the temperature changes moisture will move into or out of the pressboard via diffusion.

1.3.1 Diffusion

The rate of diffusion of moisture through the oil and the pressboard determines the time rates of change of the moisture distribution, and thus how long it takes before equilibrium is reached. Experiments have determined the diffusion constants of water in these two media to have the values shown in Table 1.1 [2].

In order to appreciate the magnitude of these diffusion constants, we can calculate that the diffusion times of water across $\Delta = 1$ mm of pressboard, given by

$$\tau = \frac{\Delta^2}{D_p} \tag{1.15}$$

Diffusion coefficient	Symbol	Value at 15°C	Value at 70°C
in oil	D_o	1.3×10^{-11} m^2/s	1.1×10^{-10} m^2/s
in pressboard	D_p	6.7×10^{-14} m^2/s	6.0×10^{-12} m^2/s

Table 1.1: Diffusion coefficients of water in transformer oil and pressboard [2]

are half a year and two days at 15°C and 70°C respectively. What that means is that equilibrium is generally never reached in an operating transformer, given how quickly the oil temperature changes with the power load and the ambient air temperature. Instead, oil equilibrates only with a thin layer of pressboard at its surface. This implies that the surface of the pressboard may become extremely dry, which could lead to static charge accumulation and partial discharges, ultimately leading to catastrophic failure.

1.3.2 Equilibrium

The equilibrium of moisture between the oil and the pressboard is what determines the direction of the mass transfer processes in the pressboard/oil system. This equilibrium is extremely sensitive to temperature, as can be seen in Figure 1-3. This is how a temperature transient drives the system out of equilibrium and initiates the mass transfer processes. If, for example, the moisture concentration in the paper is 0.5%, at 20°C, the oil humidity in equilibrium with it is about 0.5 ppm. If the oil temperature then changes to 80°C, the new equilibrium value for the oil humidity becomes close to 6.5 ppm, i.e. thirteen times higher, which would drive water out of the pressboard surface and leave it very dry until moisture deep in the pressboard diffuses to the surface on a time scale of order τ in equation 1.15.

1.4 Imposed ω-k Dielectrometry

The simplest way to measure the complex permittivity of a material is to place it between parallel electrodes and then measure its complex impedance. This is the idea behind the parallel-plate sensor described in Section 2.1. In that case the electric fields

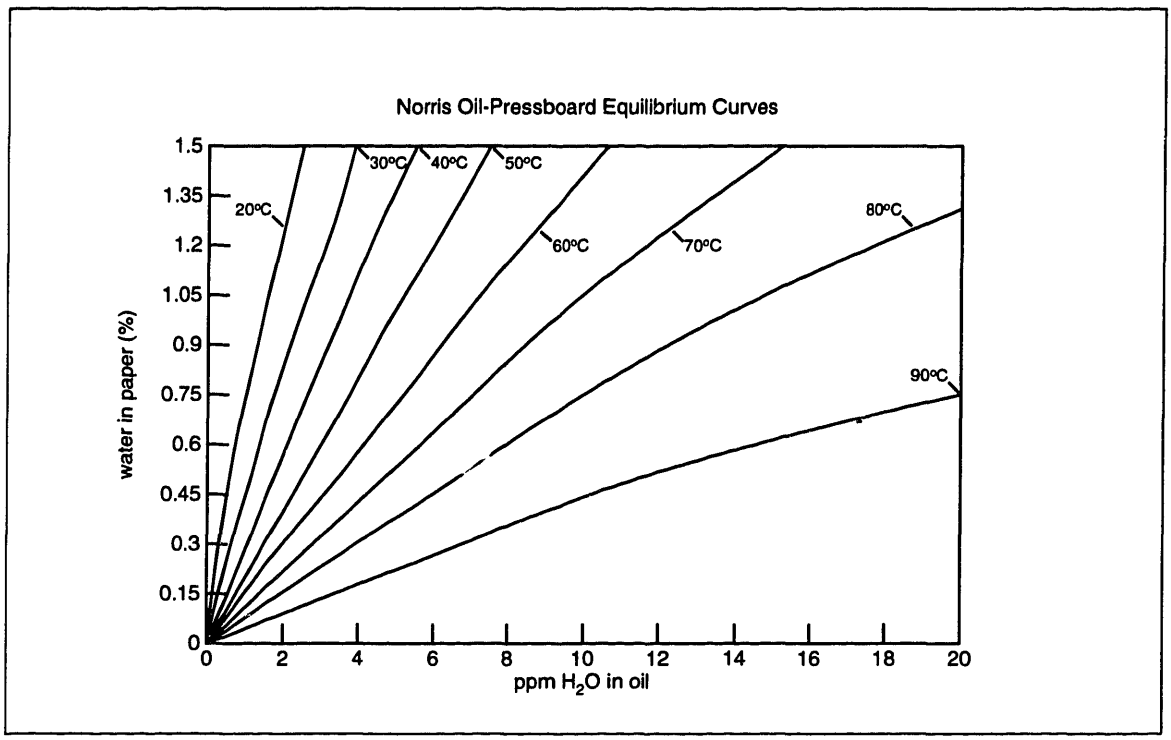

Figure 1-3: Equilibrium relationship between the moisture content of transformer oil and pressboard for temperatures ranging from 20°C to 90°C

are uniform and independent of position in space. If instead the two electrodes are placed side by side only on one surface of the material, the electric fields will decrease away from the electrodes and the complex impedance between the two electrodes will be most sensitive to the material adjacent to them. The disadvantage of this two-dimensional method is that the problem of calculating the impedance as a function of the material's complex permittivity is much more complicated.

The idea of placing both electrodes on the same surface is at the base of the method of imposed ω-k dielectrometry. The two electrodes are shaped as a multitude of interdigitated fingers, as shown in Figure 1-4. The electric fields are uniform in the z-direction and periodic in the y-direction with a spatial wavelength of λ. Thus at every surface of constant x the electric potential is periodic in y and can be expanded as an infinite series of sinusoidal Fourier modes of spatial wavelengths $\lambda_n = \lambda/n$. This is very convenient, because the solutions to Laplace's equation

$$\nabla^2 \Phi = 0 \tag{1.16}$$

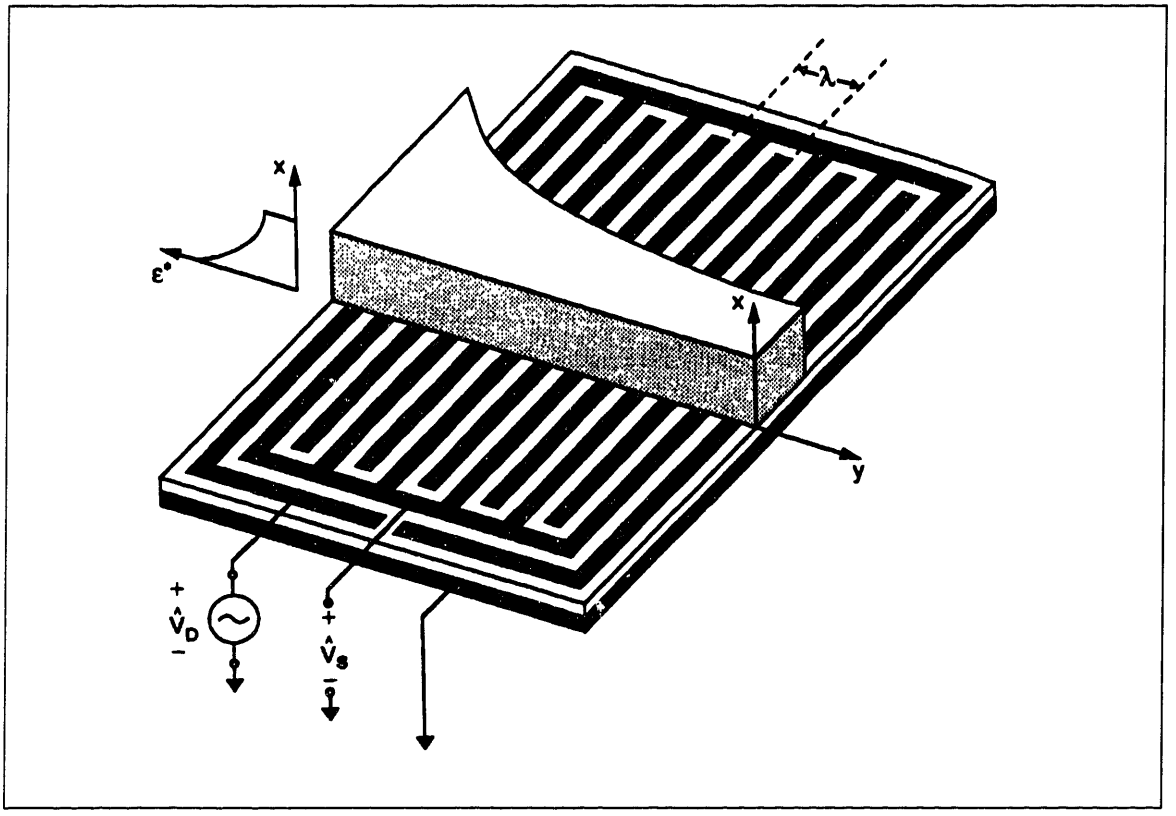

Figure 1-4: Imposed ω-k dielectrometry

in Cartesian geometry are of the form

$$\Phi = \Phi_0 \, \mathrm{hyp}(kx) \, \mathrm{trig}(ky) \tag{1.17}$$

where $\mathrm{hyp}(x)$ stands for any one of the hyperbolic exponential functions $\sinh(x)$, $\cosh(x)$, e^x, or e^{-x}, and $\mathrm{trig}(x)$ stands for one of the trigonometric functions $\sin(x)$ or $\cos(x)$. For every Fourier mode n, the electric fields decrease with x as $\exp(-2\pi nx/\lambda)$ with the fundamental mode $n = 1$ penetrating farthest into the material.

By designing sensors with various spatial wavelengths λ, it is possible to test the dielectric properties of materials at different depths. Combining the results from several such sensors makes it possible to determine the x-dependent spatial profiles of the complex permittivity.

The three-wavelength sensor, described in detail in Chapter 3, uses the ideas presented in this section. Section 3.3 in that chapter develops the mathematical

model of the interdigitated sensors.

1.5 Scope of Thesis

In this thesis we present the several stages of research that lead to the ultimate goal of studying the dynamics of mass transfer processes in pressboard/oil systems by measuring moisture profiles.

First, we establish a relationship between the moisture content of pressboard and its complex permittivity. In this way we can convert dielectric profiles into moisture profiles. Chapter 2 presents the methods used in the establishment of this relationship.

The next step is to introduce spatially dependent dielectrometry measurements, which provide information about the spatial variation of the complex permittivity. Such sensors are the interdigital sensors. Chapter 3 presents the construction and modeling of the interdigital sensors in general, with specific emphasis on the three-wavelength sensor, which is a hybrid sensor capable of taking measurements at three distinct spatial wavelengths simultaneously.

Chapter 4 discusses the various issues associated with the interpretation of data from the interdigitated sensors. It also presents in detail several numerical algorithms which are used for the interpretation of such data and the establishment of spatial profiles.

Finally, in Chapter 5 we present the results from the application of the concepts developed in the previous three chapters to actual measurements with the three-wavelength sensor. Future work may include applying the entire methodology established in this thesis to monitoring and studying of the mass transfer processes of water in a simulated transformer environment.

In addition to presenting new concepts and results from experiments and theoretical work in the subject of interdigital dielectrometry, this document is also meant to serve as a reference for those who are interested in continuing the work presented in it. Consequently, the experimental procedures and setups are presented with in great detail. We are also including a complete listing of all programming code used

in the implementation of the various numerical procedures and in the process of data acquisition and interpretation. Familiarity with references [3] and [2] would prove to be very helpful to the reader of this document.

Chapter 2

Features of the Dielectric Spectrum of Pressboard

2.1 Parallel Plate Sensor

The simplest way to measure the permittivity and the conductivity of a material is to place it between a pair of parallel plates of known area and separation, thus producing a lossy capacitor. This test cell can be modeled as a resistor in parallel with a capacitor. The complex admittance of the structure can then be measured, and from there its permittivity and conductivity can be calculated.

We have used this simple idea in the development of the *parallel-plate sensor.* Its structure is shown in Figure 2-1. The figure shows more than just a pair of conducting plates. The actual capacitive structure is comprised of the *driven electrode* and the *sensing electrode.* Underneath the sensing electrode lies the *guard electrode.* The guard electrode is driven by a buffer amplifier stage to be always at the same potential as the sensing electrode. The buffer amplifier is situated in the interface box, described in detail in Appendix E. The sensing electrode is also surrounded by a ring electrode, which is connected to the guard electrode. In addition to shielding the sensing electrode from external electric fields, the guard electrode serves to eliminate all parallel parasitic capacitances and resistances, which the sensing electrode might have with respect to the surrounding medium. Such parasitic impedances are in effect

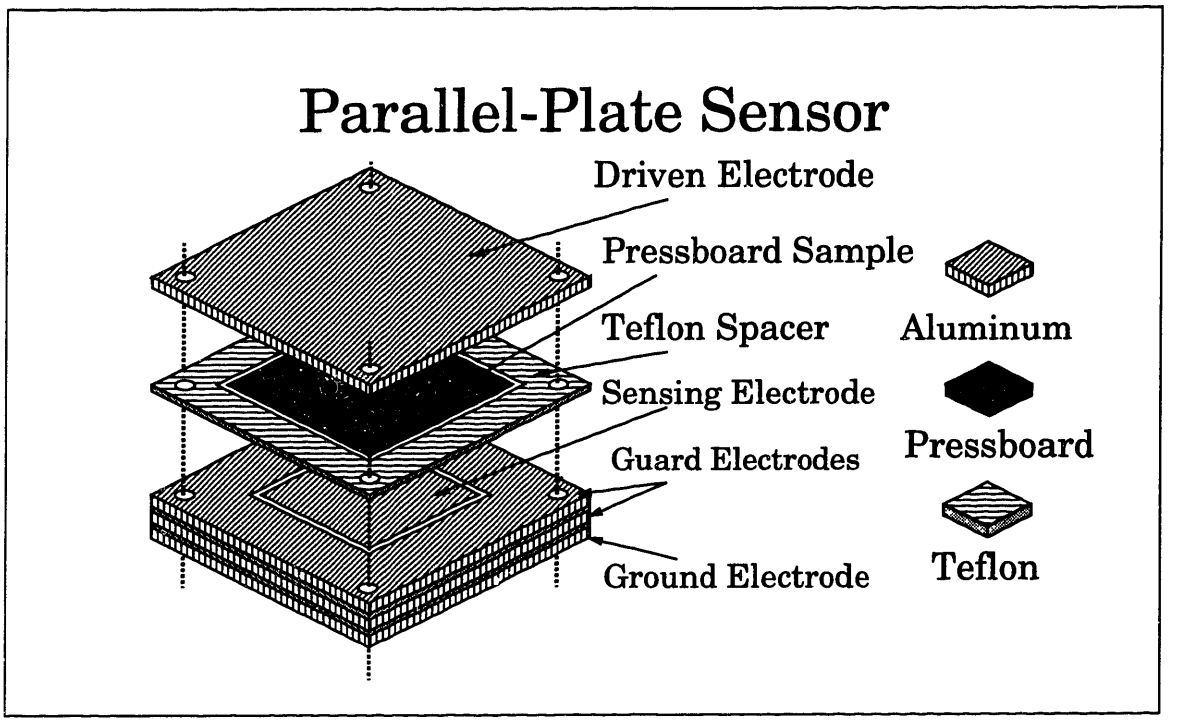

Figure 2-1: Structure of the parallel-plate sensor

multiplied by the gain of the operational amplifier, making their effects negligible.

Although the guard electrode is driven by a much lower impedance source as compared to the sensing electrode, namely the operational amplifier, it is still necessary to shield it from outside fields, and that is the purpose of the ground electrode. A triaxial cable is used to connect the sensor to the interface box. The center conductor is connected to the sensing electrode, the middle — to the guard electrode, and the outer — to ground. The ideas about shielding, as discussed above, are fully applicable to the connecting triaxial cable too. The driving voltage is applied via a separate coaxial cable.

Another advantage to having a guard ring around the sensing electrode is that the electric field is highly uniform and there are essentially no fringing fields associated with that electrode. The material sample is larger in area than the sensing electrode, thus letting all field lines terminating on it pass through the material sample. Teflon was chosen as the insulating material between the different electrodes because of its excellent thermal properties in addition to being a very good insulator. The entire

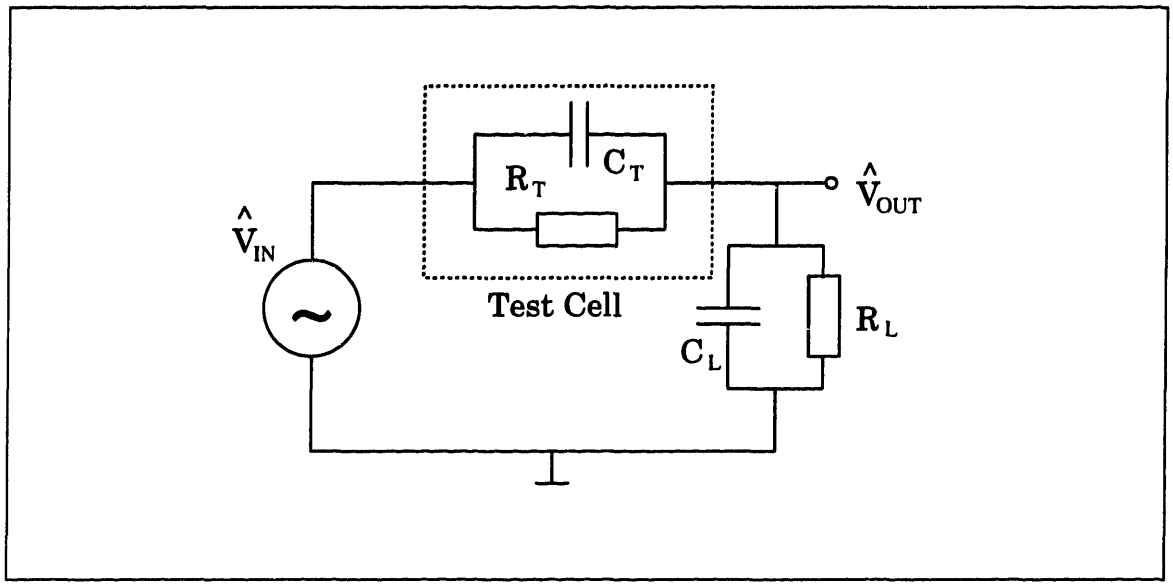

Figure 2-2: Equivalent circuit of the test structure

'sandwich' structure is tightened together with insulating nylon bolts.

2.1.1 Circuit Model

As shown in Appendix E, the input admittance of the interface box, with which the sensing electrode is loaded, is that of a known parallel RC-pair. Therefore the test structure relevant to the measurement may be modeled as shown in Figure 2-2.

The following equations relate the test-cell lumped parameters R_T and C_T (see Figure 2-2):

$$R_T = \frac{d}{\sigma A} \tag{2.1}$$

$$C_T = \frac{\epsilon A}{d} \tag{2.2}$$

where d is the plate separation distance, A is the sensing electrode area, and σ and ϵ are the material's conductivity and permittivity respectively. Equations 2.1 and 2.2 make it clear that the geometry of the test cell may be described by a single parameter,

the capacitance of the structure in air (C_{AIR}):

$$C_{AIR} = \frac{\epsilon_0 A}{d} \tag{2.3}$$

which is an easily measured parameter. In terms of equation 2.3, we have:

$$R_T = \frac{\epsilon_0}{\sigma C_{AIR}} \tag{2.4}$$

$$C_T = \frac{\epsilon C_{AIR}}{\epsilon_0} \tag{2.5}$$

For linear time-invariant (LTI) systems we take the standard form:

$$V_{IN} = \Re\{\hat{V}_{IN} e^{st}\} \tag{2.6}$$

$$V_{OUT} = \Re\{\hat{V}_{OUT} e^{st}\} \tag{2.7}$$

with $\hat{V}_{IN}$ and $\hat{V}_{OUT}$ defined in Figure 2-2. The controller, described in Appendix D is responsible for generating the driving voltage and measuring the output voltage. The data that it produces is expressed in terms of a magnitude $20\log(M)$ [**dB**] and phase $\varphi \cdot 180/\pi$ [**deg**], which are related to the complex amplitudes defined in Figure 2-2 in the following way:

$$M = \left| \frac{\hat{V}_{OUT}}{\hat{V}_{IN}} \right| \tag{2.8}$$

$$\varphi = \angle \left(\frac{\hat{V}_{OUT}}{\hat{V}_{IN}} \right) \tag{2.9}$$

Since the values of M and φ are needed as defined above, as opposed to the way they are presented by the controller (i.e. in **dB** and **deg**), they need to be transformed to that form first. The symbol $\angle$ used in equation 2.9 is defined as:

$$\angle z = \tan^{-1} \frac{\Im\{z\}}{\Re\{z\}} \tag{2.10}$$

for a complex number z.

The next step in calculating σ and ϵ is to calculate R_T and C_T from measurements of M and φ, and the known values of R_L and C_L. We define the admittances of the test and load branches as $Y_T = 1/R_T + sC_T$ and $Y_L = 1/R_L + sC_L$ respectively, where s is the complex frequency. Then from the voltage divider relationship we obtain:

$$\frac{\hat{V}_{OUT}}{\hat{V}_{IN}} = \frac{Y_T}{Y_T + Y_L} = \frac{\frac{1}{R_T} + sC_T}{\frac{1}{R_T} + \frac{1}{R_L} + s(C_T + C_L)} = \frac{C_T}{C_T + C_L} \cdot \frac{s+p}{s+z} \tag{2.11}$$

with the zero z and the pole p located at:

$$z = -\frac{1}{R_T C_T} \tag{2.12}$$

$$p = -\left(\frac{1}{R_T} + \frac{1}{R_L}\right)\left(\frac{1}{C_T + C_L}\right) \tag{2.13}$$

Depending on the values of R_T, C_T, R_L, and C_L, either of z and p may be larger than the other. In the limiting case of $s \to 0$, the voltage ratio becomes real and equal to $R_L/(R_L + R_T)$. In the other extreme, where $s \to \infty$, the voltage ratio is also real and equal to $C_T/(C_L + C_L)$.

In our work we drove the system at the sinusoidal steady state, so that $s = j\omega$. Then

$$\frac{\hat{V}_{OUT}}{\hat{V}_{IN}} = Me^{j\varphi} = \frac{Y_T}{Y_T + Y_L} \tag{2.14}$$

$$\begin{aligned} \frac{Y_T}{Y_L} &= \frac{Me^{j\varphi}}{1 - Me^{j\varphi}} = \frac{M\cos\varphi + jM\sin\varphi}{1 - M\cos\varphi - jM\sin\varphi} \\ &= \frac{(M\cos\varphi + jM\sin\varphi)(1 - M\cos\varphi + jM\sin\varphi)}{1 + M^2\cos^2\varphi - 2M\cos\varphi + M^2\sin^2\varphi} \end{aligned} \tag{2.15}$$

$$\Re\left\{\frac{Y_T}{Y_L}\right\} = \frac{M\cos\varphi(1 - M\cos\varphi) - M^2\sin^2\varphi}{1 + M^2 - 2M\cos\varphi} = \frac{M\cos\varphi - M^2}{1 + M^2 - 2M\cos\varphi} \tag{2.16}$$

$$\Im\left\{\frac{Y_T}{Y_L}\right\} = \frac{M\sin\varphi(1 - M\cos\varphi) + M^2\cos\varphi\sin\varphi}{1 + M^2 - 2M\cos\varphi} = \frac{M\sin\varphi}{1 + M^2 - 2M\cos\varphi} \tag{2.17}$$

From the definitions of Y_T and Y_L we obtain

$$\frac{1}{R_T} + j\omega C_T = \frac{Y_T}{Y_L}\left(\frac{1}{R_L} + j\omega C_L\right) \tag{2.18}$$

and from there

$$\frac{1}{R_T} = \frac{M\cos\varphi - M^2}{1 + M^2 - 2M\cos\varphi}\left(\frac{1}{R_L}\right) - \frac{M\sin\varphi}{1 + M^2 - 2M\cos\varphi}(\omega C_L) \tag{2.19}$$

$$\omega C_T = \frac{M\sin\varphi}{1 + M^2 - 2M\cos\varphi}\left(\frac{1}{R_L}\right) + \frac{M\cos\varphi - M^2}{1 + M^2 - 2M\cos\varphi}(\omega C_L) \tag{2.20}$$

$$R_T = \frac{1 + M^2 - 2M\cos\varphi}{M\cos\varphi - M^2 - \zeta M\sin\varphi} R_L \tag{2.21}$$

$$C_T = \frac{M\cos\varphi - M^2 + (1/\zeta)M\sin\varphi}{1 + M^2 - 2M\cos\varphi} C_L \tag{2.22}$$

where $\zeta \equiv \omega R_L C_L$. This concludes the final step of the process of calculating σ and ϵ of a material from gain and phase data recorded by the controller.

If it is necessary to calculate R_L and C_L based on knowledge of R_T and C_T, which occurs if we want to measure the load impedance of an interface box by replacing the test cell with a known test impedance, then the formulas take up the following form:

$$R_L = \frac{M}{\cos\varphi - M + \zeta\sin\varphi} R_T \tag{2.23}$$

$$C_L = \frac{\cos\varphi - M - (1/\zeta)\sin\varphi}{M} C_T \tag{2.24}$$

which is particularly useful for diagnostics of interface boxes (see Appendix E). The program **testrc.c** uses these formulas.

This inversion process is carried out by the program **inv.c**, listed in Section G.4, which takes as an input the raw output file generated by the controller box, and outputs values for $\epsilon' = \epsilon$ and $\epsilon'' = \sigma/\omega$ in files with extensions **.e1** and **.e2** respectively. The program reads the setup file **.invsetup**, also listed in Section G.4, which contains the default values for C_{AIR}, C_L, and R_L. An alternative setup file may be given as

an argument.

2.1.2 Testing

In order to test the performance of the parallel-plate sensor, we used it on known materials, in particular Teflon and transformer oil. Figure 2-3 shows the gain and phase of the measurement on Teflon. Only ϵ/ϵ_0 is shown, because σ was too low to measure. In the figure the average measured value of ϵ, the relative dielectric constant, is $\epsilon/\epsilon_0 = 2.1$, which is exactly the value quoted in the literature [4]. There is no variation with frequency over the range of 0.005–10,000 Hz, which is consistent with the known non-dispersive properties of Teflon.

Shell Diala A transformer oil was used in the oil experiment. Figure 2-4 shows the results. On a log-log scale, the plot of ϵ'' versus frequency is a straight line of slope -1, which means that σ is independent of the frequency. This is characteristic of an *ohmic* material. For linear dielectric materials, ϵ' should also be constant with frequency. The observed rise of ϵ' at the lower end of the frequency range can be attributed to double layer formation at the aluminum-oil interface [2]. This plot corresponds to $\sigma = 0.83 \times 10^{-12}$ $\mho$/m and $\epsilon/\epsilon_0 = 2.2$, which are typical values for the dielectric parameters of this kind of transformer oil.

The plot of ϵ'' is not shown for frequencies higher than $10^{0.7}$ Hz. This is because at that frequency range the response is fully dominated by the capacitive element, and no meaningful information about the conductivity may be inferred. This insensitivity of the response to the conductivity is discussed in more detail the next subsection.

2.1.3 Measurement Sensitivities to the Load Impedance

Looking at the circuit in Figure 2-2 it is immediately obvious that when $\omega \to \infty$ and $\omega \to 0$ the response will be fully dominated by the capacitors or the resistors respectively. In those two extreme cases the complex amplitude ratio is purely real, corresponding to a phase angle of zero. Looked at from another angle, if $\varphi = 0$, then one of the two equations 2.21 and 2.22 will yield incorrect results, depending on at

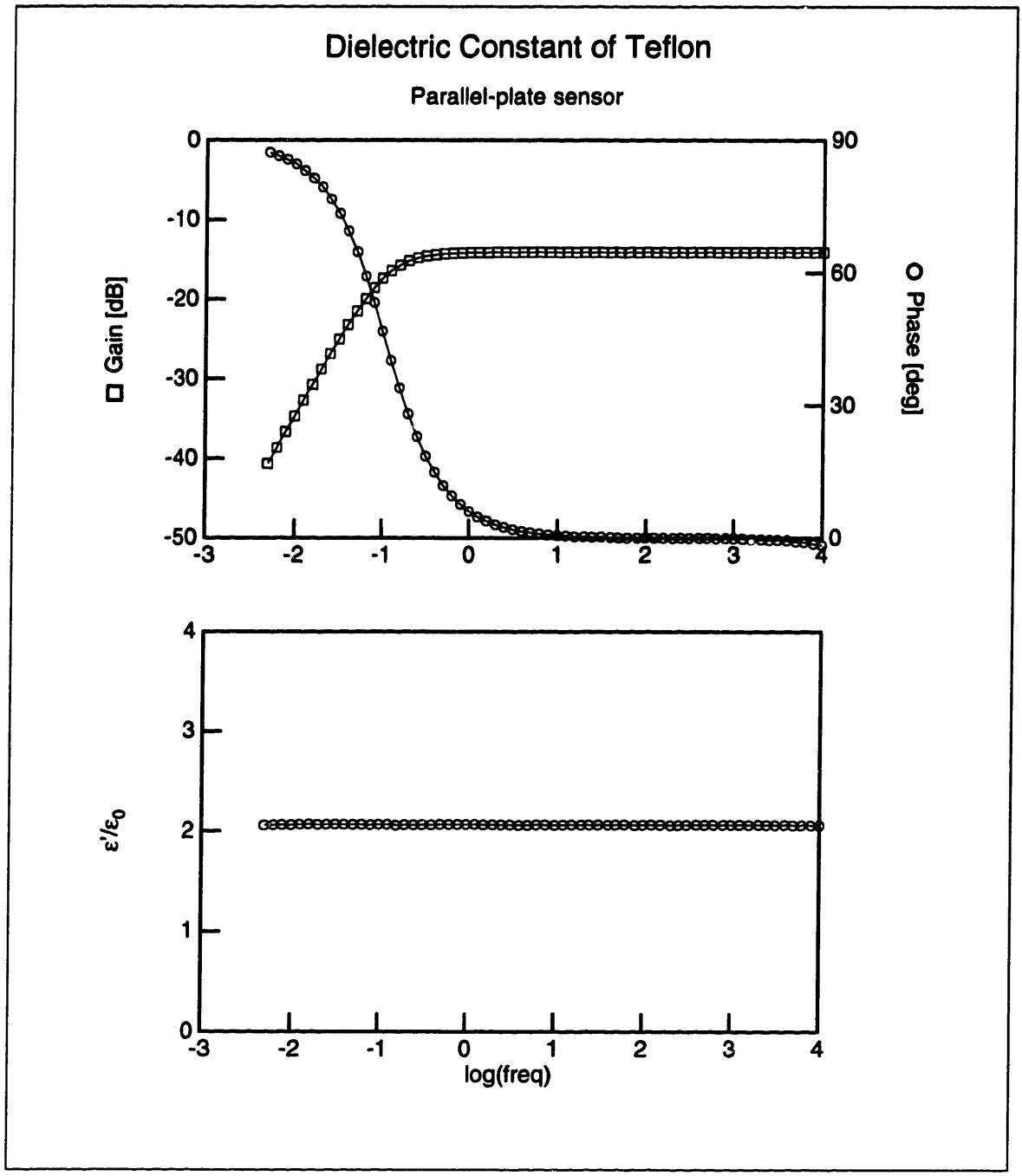

Figure 2-3: Relative dielectric constant of Teflon measured with the parallel-plate sensor

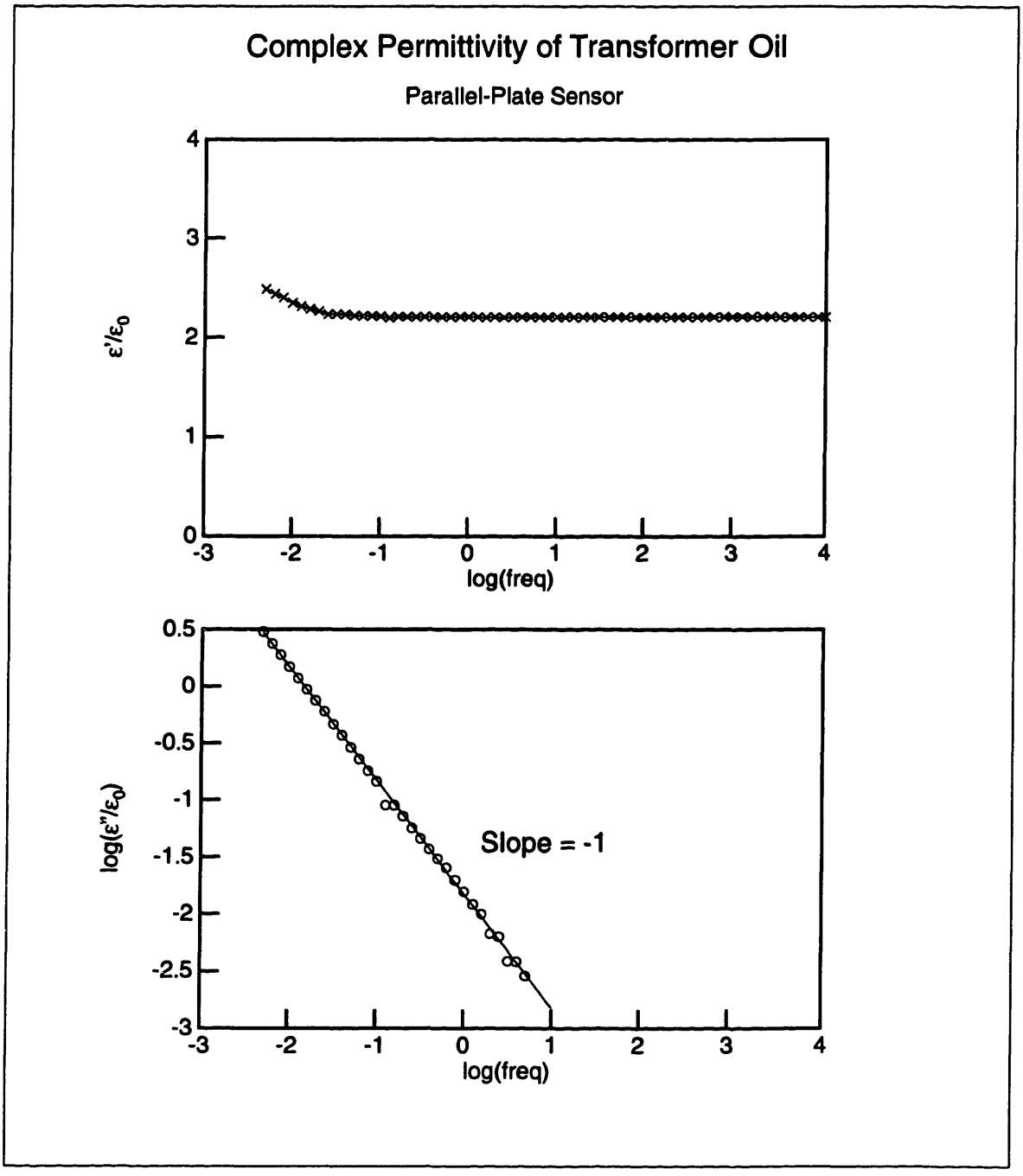

Figure 2-4: Complex permittivity of transformer oil measured with the parallel-plate sensor

which extreme the frequency is. There is a special case, namely $R_T/R_L = C_L/C_T$, when $\varphi = 0$ for every ω. From a strictly mathematical point of view, this special case is the only case when φ is exactly zero. This is why these equations "assume" when given $\varphi = 0$ that this special case holds.

In reality, of course, we are limited by the precision of the equipment, and therefore it is impossible to measure resistances reliably above a certain frequency. It is similarly impossible to obtain reliable estimates for the capacitances below a certain frequency. Since we have some flexibility in choosing R_L and C_L, we should like to choose such values that our measurements of R_T and C_T would be most reliable. This is the topic of this section.

Equations 2.12 and 2.13 define the zero and the pole of the system, which roughly delimit the interval of frequencies for which both R_T and C_T can be reliably estimated. Somewhere between these two frequencies, φ reaches an extremum, before returning to zero again (see Figure 2-5). If the pole frequency is lower than the zero frequency, φ is always negative, and if the pole is at a higher frequency, than the zero φ is always positive. We would like to place the peak (or trough) of φ close to the center of the interval of frequencies in which we are most interested. This is how we came up with the values of $R_L = 9.8$ GΩ and $C_L = 120$ pF shown in Figure E-1.

So far the discussion of sensitivity has been qualitative. In order to quantify these considerations, we go on to calculate the sensitivities of the estimated values for R_T and C_T. We define the *sensitivity* of a quantity y with respect to a quantity x as follows:

$$S_x^y = \frac{1}{y} \cdot \frac{\partial y}{\partial x} \tag{2.25}$$

The sensitivity describes what the relative change in y would be for a change in x. If G and p are the magnitude and phase of the response expressed in **dB** and **deg** respectively, related to M and φ as follows,

$$G = 20 \log M \tag{2.26}$$

$$p = \frac{180}{\pi}\varphi \tag{2.27}$$

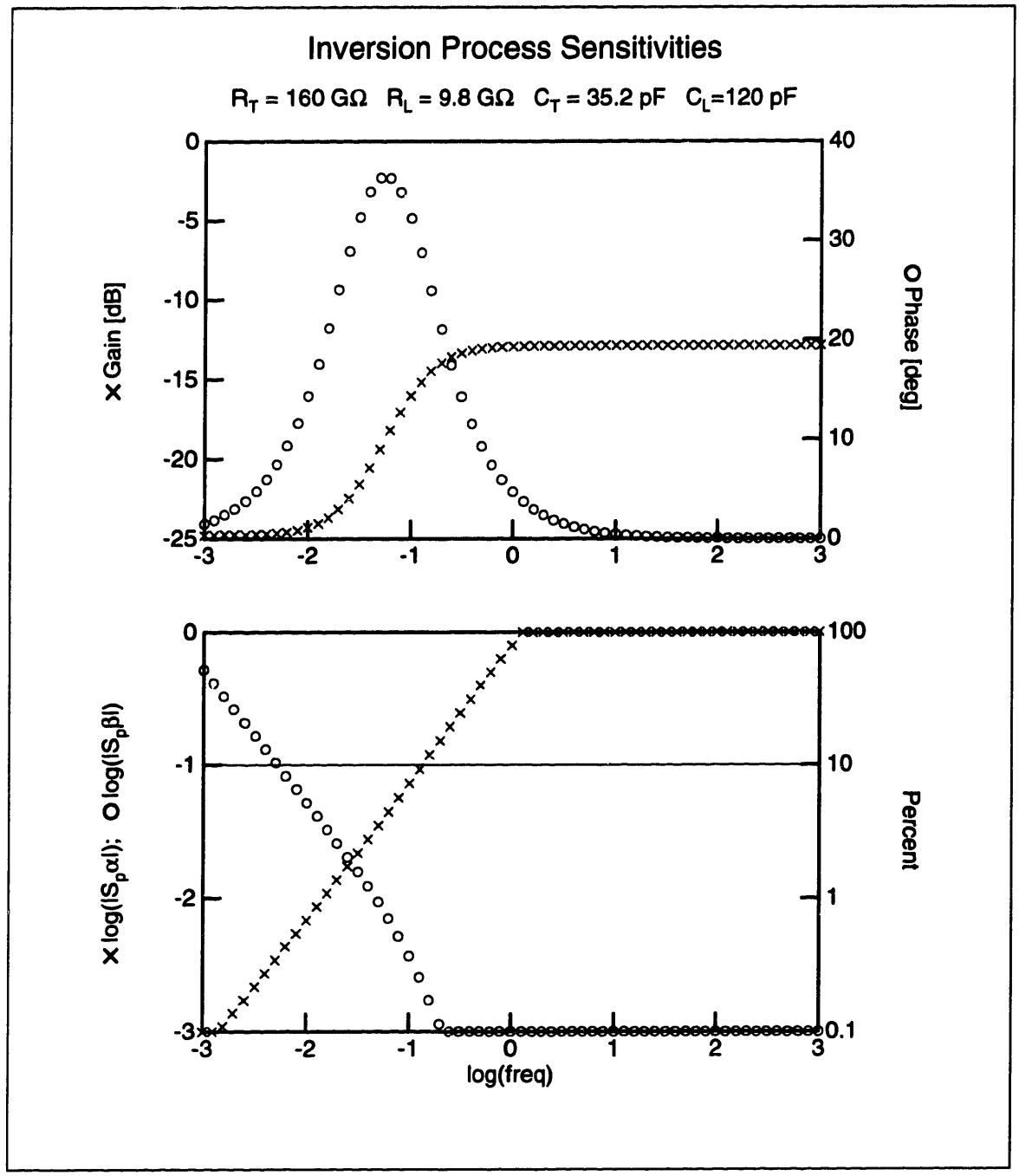

Figure 2-5: Sensitivity of the inversion formulas to noise. The bottom plot shows the dependence of the logarithm of S_p^α and S_p^β with frequency. These sensitivities show how much α and β would change for a small change in p. These ratios are least sensitive to noise for low values of S_p^α and S_p^β

and we make the additional definitions

$$\alpha \equiv \frac{R_L}{R_T} = \frac{M\cos\varphi - M^2 - \zeta M\sin\varphi}{1 + M^2 - 2M\cos\varphi} \tag{2.28}$$

$$\beta \equiv \frac{C_T}{C_L} = \frac{M\cos\varphi - M^2 + (1/\zeta)M\sin\varphi}{1 + M^2 - 2M\cos\varphi} \tag{2.29}$$

then we obtain the following equations for the sensitivities of α and β with respect to G and p:

$$S_G^\alpha = \frac{\partial\alpha}{\partial M}\cdot\frac{dM}{dG} = \frac{(1+M^2)\cos\varphi - 2M - \zeta(1-M^2)\sin\varphi}{(1+M^2-2M\cos\varphi)^2}\cdot\frac{M\ln 10}{20\alpha} \tag{2.30}$$

$$S_p^\alpha = \frac{\partial\alpha}{\partial\varphi}\cdot\frac{d\varphi}{dp} = \frac{M\{(M^2-1)\sin\varphi - \zeta[(M^2+1)\cos\varphi - 2M]\}}{(1+M^2-2M\cos\varphi)^2}\cdot\frac{\pi}{180\alpha} \tag{2.31}$$

$$S_G^\beta = \frac{\partial\beta}{\partial M}\cdot\frac{dM}{dG} = \frac{(1+M^2)\cos\varphi - 2M + (1/\zeta)(1-M^2)\sin\varphi}{(1+M^2-2M\cos\varphi)^2}\cdot\frac{M\ln 10}{20\beta} \tag{2.32}$$

$$S_p^\beta = \frac{\partial\beta}{\partial\varphi}\cdot\frac{d\varphi}{dp} = \frac{M\{(M^2-1)\sin\varphi + (1/\zeta)[(M^2+1)\cos\varphi - 2M]\}}{(1+M^2-2M\cos\varphi)^2}\cdot\frac{\pi}{180\beta} \tag{2.33}$$

Figure 2-5 shows gain G and phase p as a function of frequency. It also shows $\log|S_p^\alpha|$ and $\log|S_p^\beta|$. One can see that α and β, and consequently R_T and C_T, are least sensitive to noise in the vicinity of the extremum of phase.

2.2 Experimental Procedures

The objective of this set of experiments is to study how the dielectric spectrum of oil-impregnated pressboard changes with variations in temperature and moisture content. The dielectric spectrum is to be measured with the parallel-plate sensor, described in detail in Section 2.1.

We first prepared many samples of pressboard, each with a different content of water. We then measured the moisture content of a sample, placed it in the sensor structure, scanned its dielectric spectrum at five different temperatures and finally measured its moisture content again. This section describes all of these stages.

Sample Name	Vacuum Drying			Oil Immersion	
	Temperature [°C]	Duration [hours]	Vacuum [mTorr]	Temperature [°C]	Duration [minutes]
NB	70	12		25	5×10^5
ND	70	24		70	60
MA	70	10	300	70	10
MB	70	2/3	400	60	10
MC	70	1/3	550	70	10
MD	70	2	200	70	10
MF	70	4	160	70	10
MG	70	15.5	100	70	10

Table 2.1: Impregnation process parameters for the pressboard samples used in the universal spectrum

2.2.1 Impregnation

The equipment used to impregnate our samples of pressboard with transformer oil is described in detail in Appendix C. Prior to impregnation we cut 50 mm×50 mm pieces of 1 mm thick oil-free EHV-Weidmann HIVAL pressboard. Then we placed them in the impregnation chamber, one at a time, for various lengths of time, in order to obtain different moisture contents. Table 2.1 lists the parameters of the oil-impregnation process that every sample underwent.

2.2.2 Moisture Measurements

The moisture of each sample was measured before and after it was placed in the parallel-plate sensor with the help of the Mitsubishi VA-05 water vaporizer and the Mitsubishi CA-05 moisture meter. The use of this equipment is described in Appendix B. That appendix also discusses the need to split the pressboard samples into many thin layers before depositing in the vaporizer oven, a procedure strictly followed in this set of measurements.

We define the moisture content of pressboard as the weight of water liberated from the sample during vaporization (a quantity provided by the moisture meter) divided by the total weight of the oil-impregnated sample before it is placed in the

%	NB	ND	MA	MB	MC	MD	MF	MG
Moisture	3.1	1.1	2.3	1.8	2.2	0.42	0.83	1.8

Table 2.2: Moisture Measurements for the pressboard samples used in the universal spectrum

oven. Since this kind of moisture measurement was destructive, in that the sample cannot be used after it has been in the vaporizer, in order to measure the moisture content of a pressboard sample, we cut off small pieces of it.

If the two moisture measurements were not close to each other, the data of the sample was not used. This happened for samples NA, NC, and ME. Table 2.2 lists the average results of the moisture measurements.

2.2.3 Temperature Transients and Control

Measurements with every sample were taken at five temperatures: 30°C, 40°C, 50°C, 60°C, and 70°C. The parallel-plate sensor, with the pressboard sample placed inside it, is placed in an oven, whose temperature is controlled by a feedback temperature controller. We could not go above 70°C because of the temperature limitations of the connecting triaxial cable. A small fan inside the oven made sure that the air is well stirred, so that there would be no temperature gradients inside the volume.

Following a change in the temperature setting, the oven temperature undergoes a transient, whose characteristics are determined by the temperature controller parameters and the thermal inertia of the oven. The temperature of the sample itself lags a bit behind the temperature of the oven. In order to determine when the sample has reached the required temperature, we measured the complex impedance of a sample at a single frequency (in order to make the measurement time short) about ten times an hour for four hours after stepping the oven setting from 25°C to 50°C. We have lost record of the frequency at which this measurement was performed, although it lies in the range 0.01–0.1 Hz. This poses no problem, since the only significance of the frequency is to ensure that ϵ'' can be reliably measured. The results from this measurement are shown in Figure 2-6. The high measured values of ϵ' and ϵ'' are due

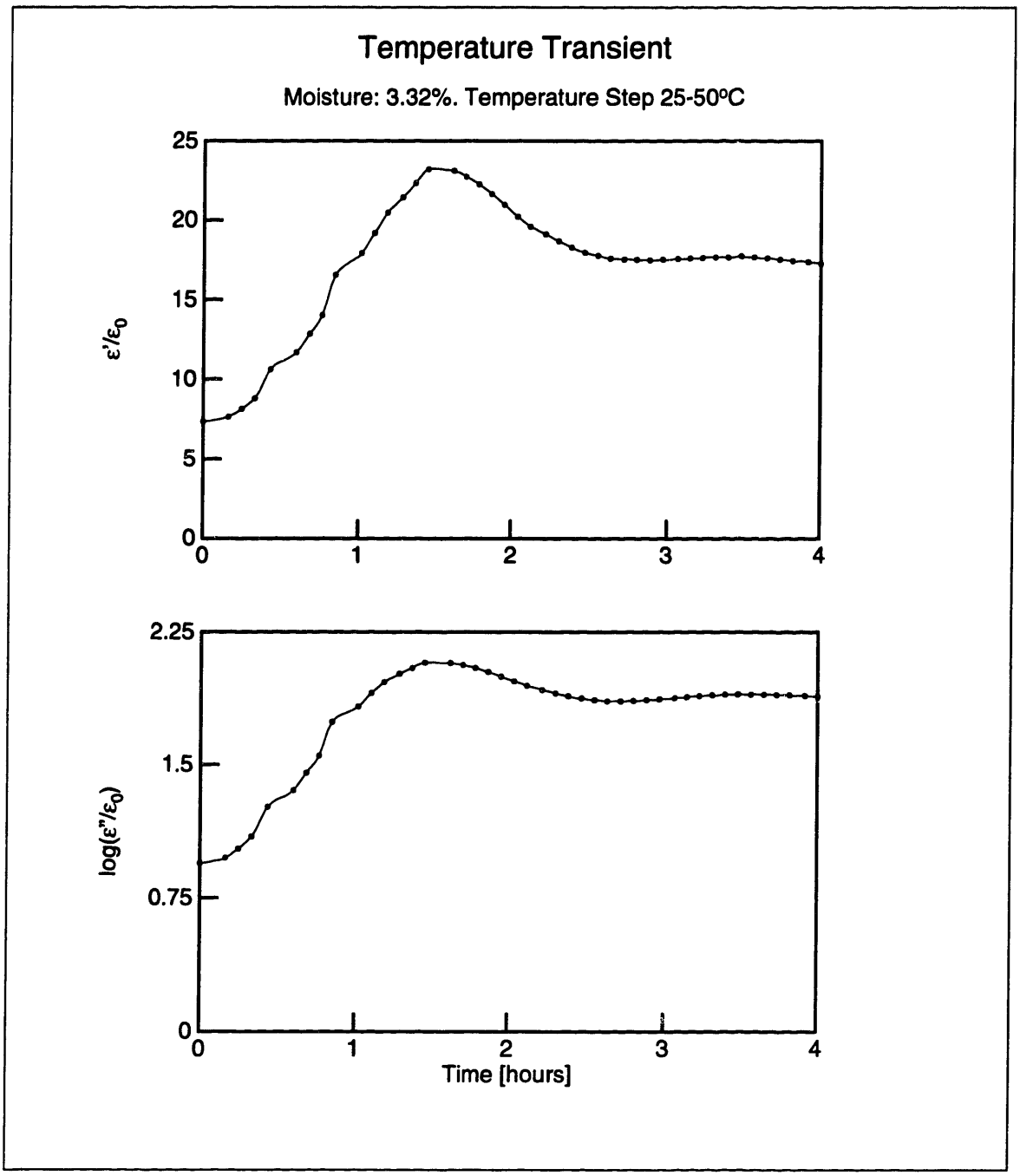

Figure 2-6: Transient in complex permittivity of a pressboard sample in response to a step in the temperature setting

to low frequency dispersion. We concluded that we must wait for about four hours after we change the temperature setting before taking a frequency scan. The high values of ϵ' are due to low-frequency dispersion in pressboard (see Section 2.3).

2.2.4 Conditioning

We have observed that in addition to the short (4 hours) temperature transient, the complex permittivity of a sample experiences another, long transient. When we tested a sample for 270 hours at a constant temperature (50°C) we observed the behavior illustrated in Figure 2-7. The long time constant of this transient suggested that it may be due to mass transfer processes of water in the pressboard. Since the sample in the test cell is sealed from the outside air, and since diffusion of water through 6 mm of pressboard before it reaches the active area would require months[1], we concluded that this *sample conditioning process* is probably due to moisture redistribution within the bulk of the pressboard, finally resulting in a uniform distribution.

We then established the rule that after a sample is impregnated and placed in the test cell, we must let it stay there for at least five days (120 hours) before any measurements are performed. This period of time for the sample to reach moisture equilibrium is necessary only once. Once it expires, only the four hours discussed in the previous section are required for the sample to reach thermal equilibrium after a temperature setting change.

2.3 Results

We would like to establish a relationship between the temperature and moisture content of pressboard, and its dielectric spectrum. This can be accomplished by

[1]Based on values for the diffusion constant taken from [2, Table 5.3], namely $D_p = 5.8 \times 10^{-12}\text{m}^2/\text{s}$ at 70°C and $6.3 \times 10^{-14}\text{m}^2/\text{s}$ at 15°C.

$$\tau = \frac{d^2}{D_p} = 36 \text{ days to } 18 \text{ years}$$

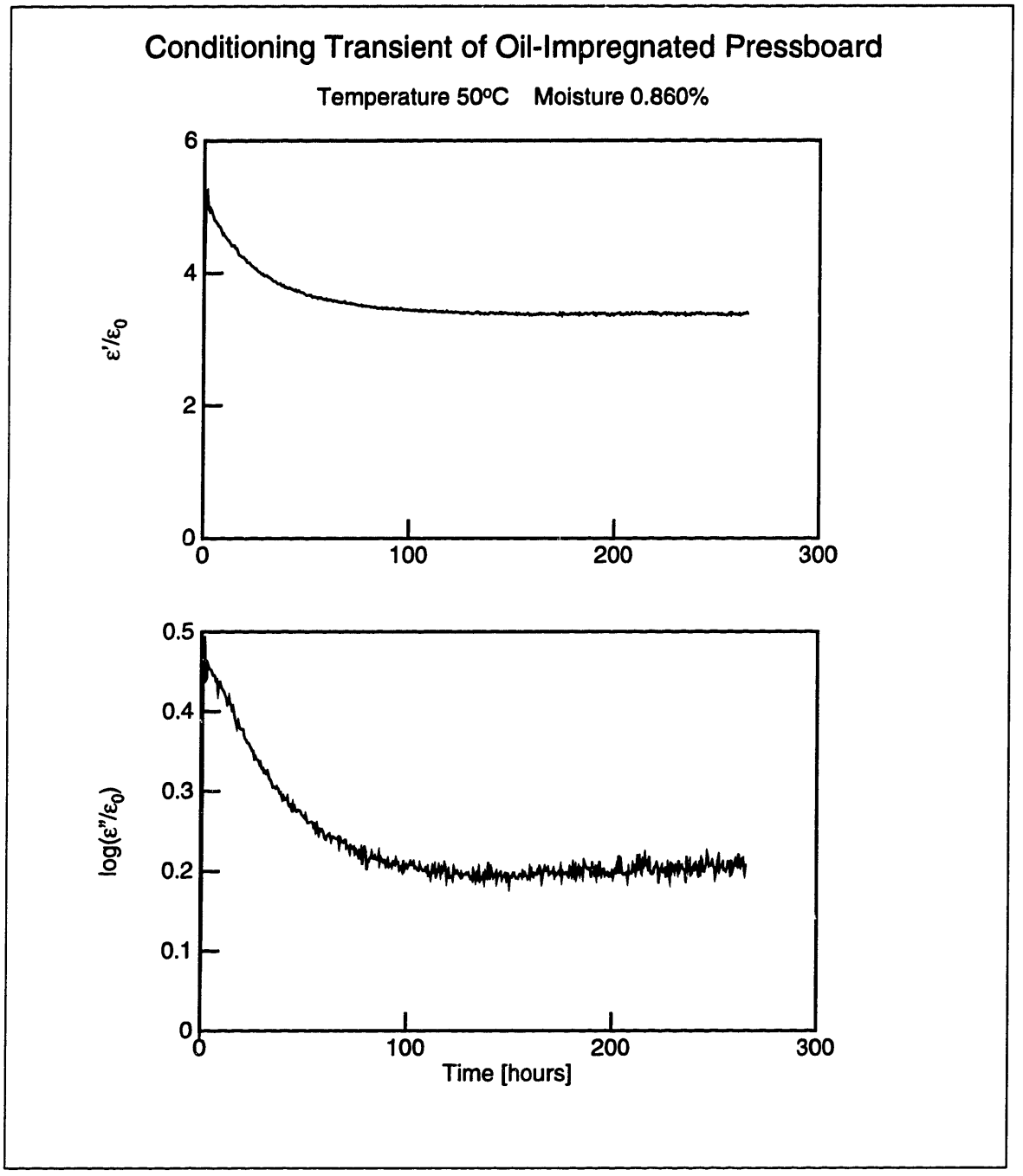

Figure 2-7: Pressboard conditioning transient

summarizing the results from frequency scans taken at several different moisture contents and temperatures.

2.3.1 Features of a Representative Dielectric Spectrum

Figure 2-8 shows the raw gain-phase data of a frequency scan of an oil-impregnated pressboard sample taken with the parallel-plate sensor. The offset data serves to check whether an unreasonably high voltage has built up at the input of the operational amplifier due to leakage currents, which could cause amplifier saturation. The measured gain and phase curves show a lot of similarity with the computer-generated ones in Figure 2-5. There are, however, some differences: One can see in Figure 2-8 that the breakpoint of the voltage ratio magnitude is at approximately $10^{-0.8} = 0.16$ Hz. This breakpoint occurs **3dB** up from the pole defined in equation 2.13, which for our experiment is to the right of the zero. Past the pole, as $\omega \rightarrow \infty$, the gain continues to change (it decays with a very slight negative slope), which is not the case in Figure 2-5. This is because the permittivity and conductivity of pressboard change with frequency, while the computer-generated data assumed constant R_T and C_T. This difference is due to the dispersive nature of pressboard which alters the shape of the curves somewhat. An ohmic material would manifest behavior similar to that in Figure 2-5.

The dispersive nature of the pressboard does not affect the validity of Equations 2.21 and 2.22, since they are evaluated at a single frequency. If we process the data shown in Figure 2-8 to produce values for the complex permittivity, we obtain the results shown in Figure 2-9. This processing of data is done with the help of the program **inv.c**, listed in Section G.4.

The first thing to note in Figure 2-9 is that all ϵ'' data for frequencies above about 10 Hz is noise. As explained in Section 2.1.3, this is due to the lack of sensitivity at high frequencies of the measurement to the resistive component of the material. When we disregard this data, the rest of the ϵ'' points lie approximately on a straight line. This line does not have a slope of -1, characteristic of an ohmic material. Instead, the slope is approximately -0.7. This comes to confirm the previous observation that

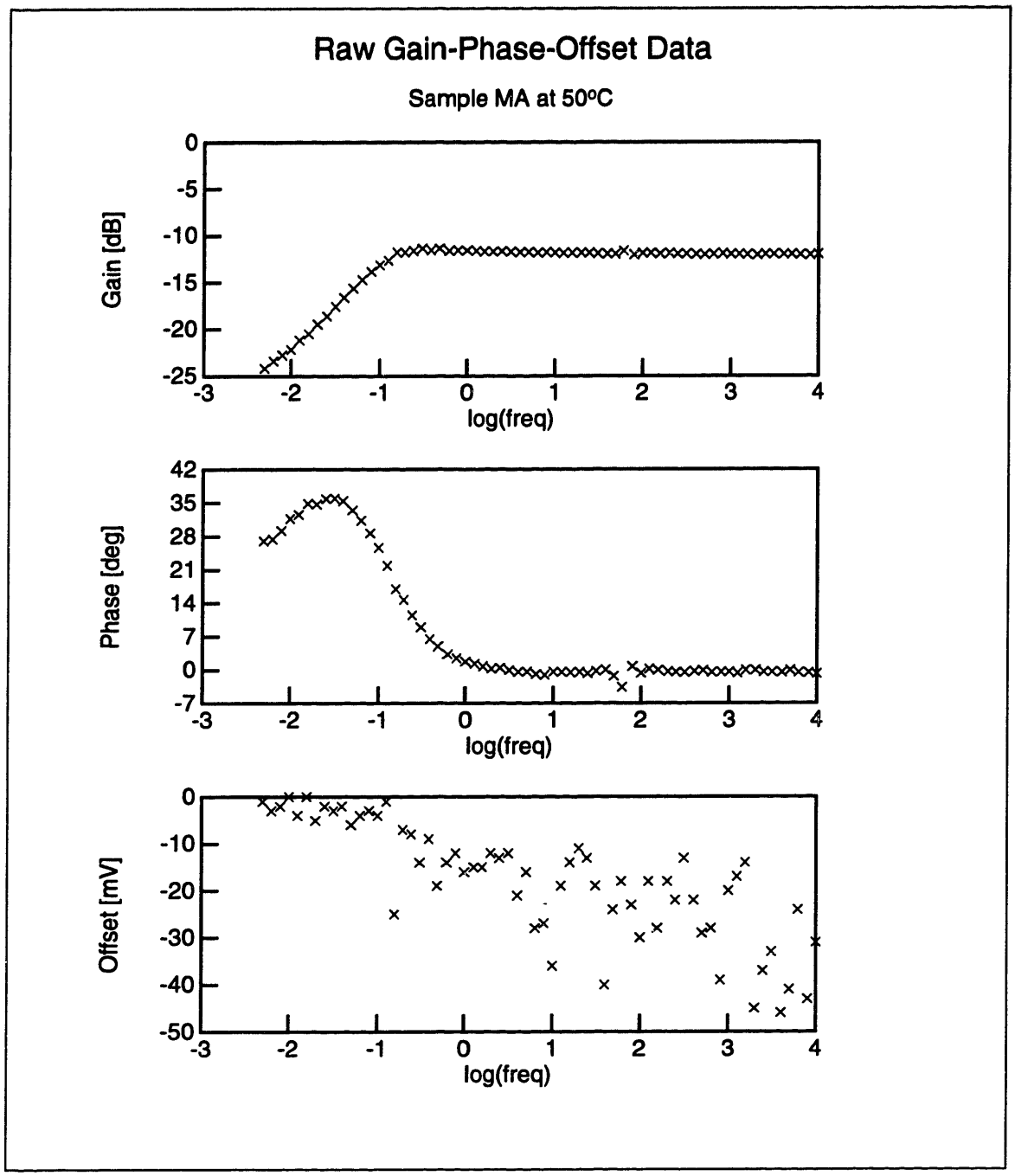

Figure 2-8: Raw gain-phase data for a frequency scan of a representative pressboard sample

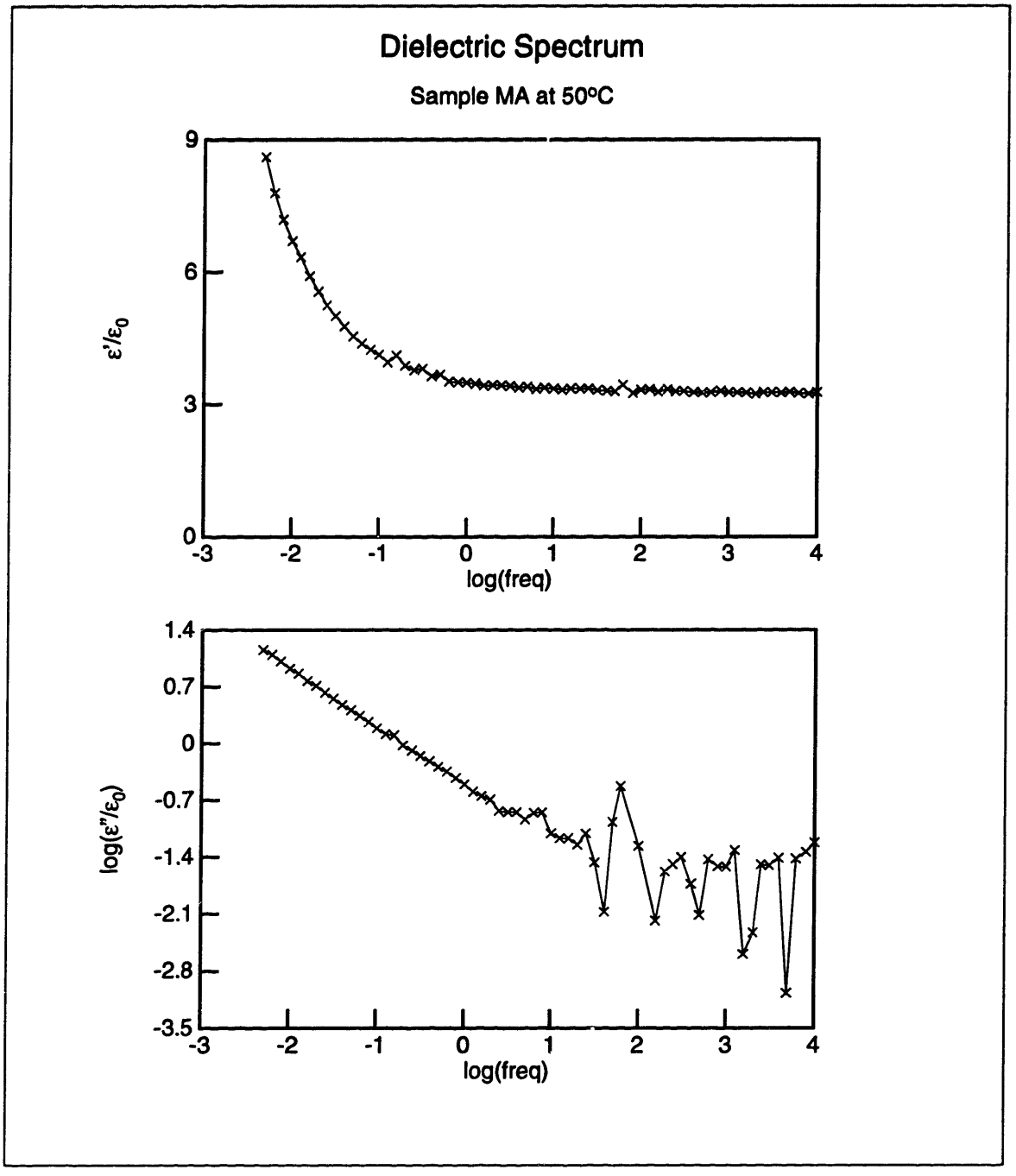

Figure 2-9: Dielectric spectrum of a representative pressboard sample

the material is dispersive.

This decay of ϵ'' is associated with a loss peak, as described in Section 1.2.2. However, the actual peak is not visible in Figure 2-9, because it occurs at a frequency which is below our bottom limit (0.005 Hz). The elevation in ϵ', which accompanies a loss peak in ϵ'' (see Section 1.2.2), is clearly shown at the top of the figure.

All but one of the pressboard samples studied displayed very similar behavior. One sample, NB, which had the highest moisture content (3.1%) is a bit different. Its dielectric spectrum is shown in Figure 2-12 and discussed in Subsection 2.3.2.

2.3.2 Frequency Shift Algorithm

Often the shape of the loss peaks in the dielectric spectrum of a material are independent of moisture and temperature. They only shift position. It should therefore be possible to create a single universal spectrum, to which all other spectra map, after having been shifted (horizontally with frequency and/or vertically) by an amount which is a function of the temperature and moisture content [5] [6].

In this case, if there is only one loss peak, the entire spectrum could be described by the position of a single point, namely the peak itself, with coordinates (ω_p, ϵ_p^*). If there are two or more peaks, and their relative position does not change (which is required if the shape is to remain constant), then a point of inflection could be chosen as the reference point [6].

Appendix A proves that a shift in either ϵ' or ϵ'', both horizontally and vertically, must be accompanied by an identical shift in the other component of ϵ^*. This is required by the Kramers-Krönig Relations (Section 1.2.2). A linear scale for ϵ' is chosen in Figure 2-9 for reasons of clarity. If, however, ϵ_∞ (the permittivity at infinite frequency) were to be subtracted from ϵ', then plotted on a log-log scale ϵ' would also be a straight line with the same slope as ϵ''. See Appendix A for a discussion of this corollary of the Kramers-Krönig Relations.

It is unfortunate that the loss peak occurs at such low frequencies, because a degree of freedom is lost by having only a straight line to shift. In other words, ω_p and ϵ_p^* cannot be determined uniquely. We have therefore the freedom of choosing to

shift the spectra either only horizontally, only vertically, or in some combination. We have chosen to move only horizontally, as suggested by research done elsewhere [6].

Since these shifts are relative, any spectrum may be chosen as the reference. The amount of shifting required to map a spectrum to the reference should be determined by some "best-fit" rule, such as a least-squares fit. If we need to find a best fit of a function $f(p_1, p_2, \ldots, p_n, x)$, where p_i are the unknown parameters, to a reference function $g(x)$ over an interval $x \in [a, b]$ by the least-squares method, we must first find the error function:

$$e(p_1, p_2, \ldots, p_n) = \int_a^b [f(p_1, p_2, \ldots, p_n, x) - g(x)]^2 dx \tag{2.34}$$

and then solve the system of n simultaneous equations:

$$\frac{\partial e}{\partial p_i} = 0, \qquad \text{for } i = 1, 2, \ldots, n \tag{2.35}$$

However, fitting straight lines presents the difficulty that the slope is already known and there is only one unknown parameter, the intercept. If the slopes are slightly different, then the two lines will not overlap perfectly and there will be no best fit on an interval of $(-\infty, \infty)$, because the integral in equation 2.34 does not exist. On a closed interval the method outlined above will place the line in a way that it crosses the other line close to the midpoint of the interval, but we do not consider this fit to be the "best fit" of a line to another line.

For these reasons we have chosen a numerical method, implemented in the program **fith.c** (Appendix G). It attempts to fit the two spectra by trying shifts in increments of 0.1 (on a logarithmic scale), because this is the frequency resolution of the controller (see Appendix D). It numerically finds the shift that minimizes the sum of the squares of the differences between the corresponding points. The results of the application of this algorithm to the data collected with the parallel-plate sensor are discussed in the next subsection.

2.3.3 Universal Spectrum

First, let us look at the spectra of the same sample at different temperatures. Figure 2-10 is a plot of all five spectra of sample MA (see Table 2.1) on the same scale for comparison. We have chosen sample MA at 50°C to be our reference spectrum. Now if we shift the other four spectra in Figure 2-10 by the appropriate amount calculated by **fith.c**, we obtain the universal curve for this sample shown in Figure 2-11.

Before we go on to integrating the results from all measurements, let us look at one particular sample, which has been excluded from consideration in all subsequent discussion. This is sample NB, whose spectra are shown in Figure 2-12. It is the sample with the highest moisture content (3.1%). Its spectra are distinctly different from those of the other seven samples. If we look at the plot of ϵ'', we can see that there are two distinct slopes. This implies that we can see the effects of *two* loss peaks, each with a different slope of decay. The one on the left is higher than the other and sufficiently close to it that the actual peak lobe of the second peak is not visible. The existence of two peaks is confirmed by the plot of ϵ', where we see a rise due to the second peak, a leveling out, corresponding to the region between peaks, and another rise associated with the first peak (see Section 1.2.2). The presence of the second peak implies either that over the extremely long process of impregnation of sample NB (about 12 months) some kind of impurity has found its way into the pressboard, or that at higher moisture levels water exists in the pressboard in a different band state.

The next step we took was to collapse (temperature only) the five spectra of every sample into universal curves, but not to try to overlap these into a single curve yet to account for moisture differences. The results are shown in Figure 2-13. One can see the seven distinct families of curves in this figure. The figure implies that these universal spectra can now be shifted again to compensate for the moisture differences and to yield the final master universal spectrum. It is shown in Figure 2-14, which contains data from 35 different frequency scans.

At the high-frequency end, the ϵ'' plots in Figure 2-14 show some spread, which is due to the high sensitivity of noise at these frequencies, already discussed in Sec-

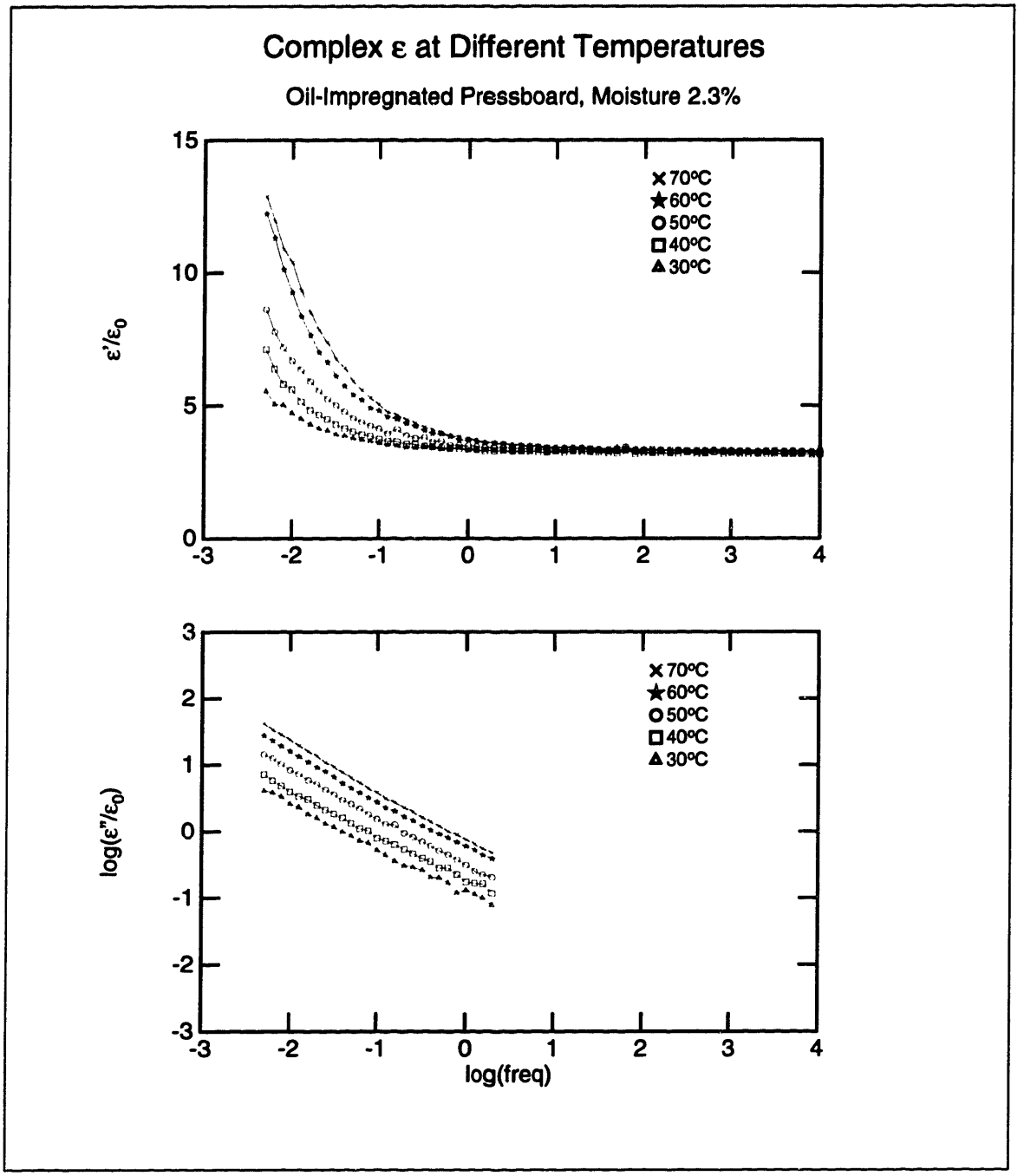

Figure 2-10: Dielectric spectra of a pressboard sample (MA) at five temperatures

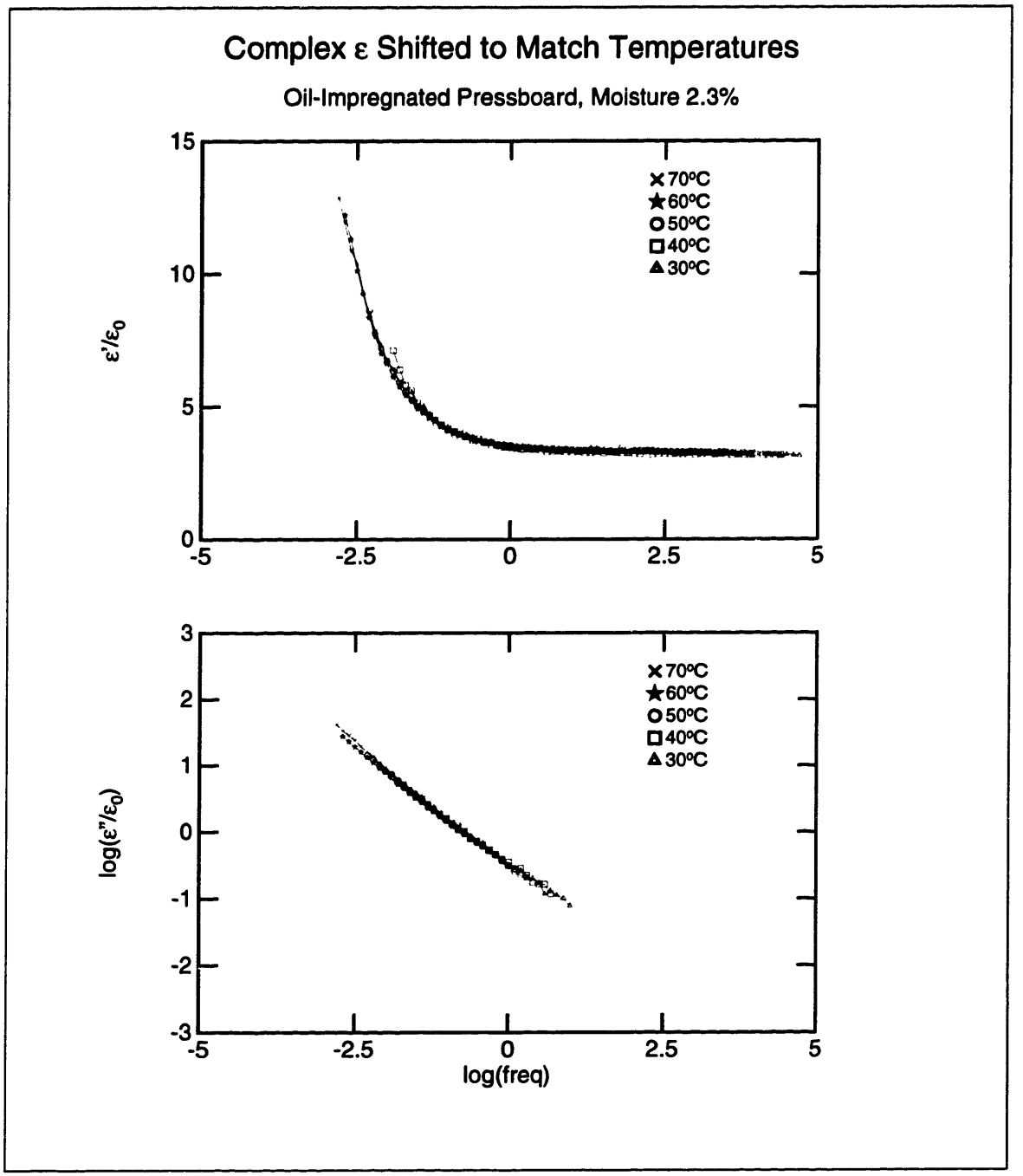

Figure 2-11: Universal curve for one sample (MA) at five temperatures

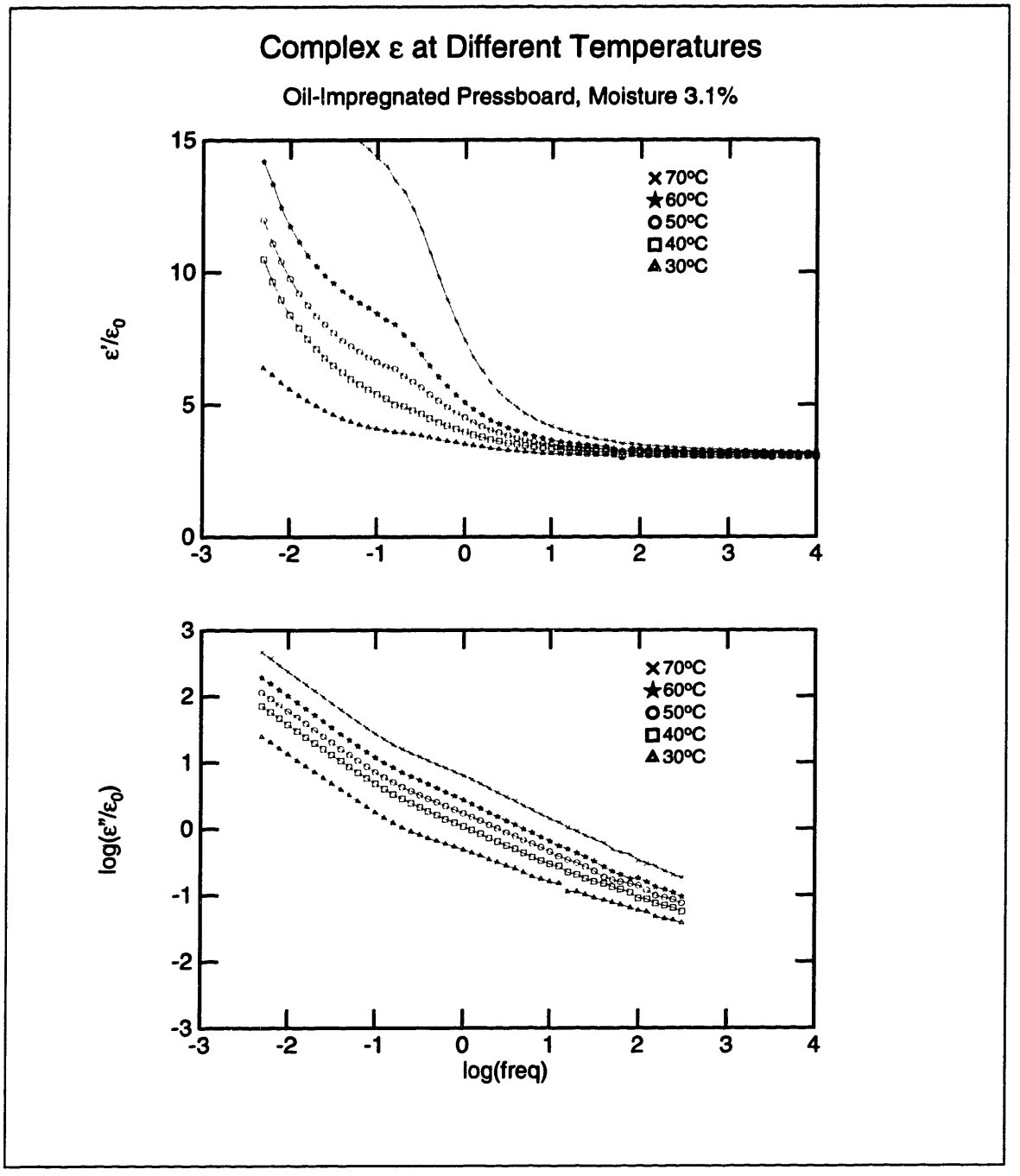

Figure 2-12: Dielectric spectra of a high water content pressboard sample (NB)

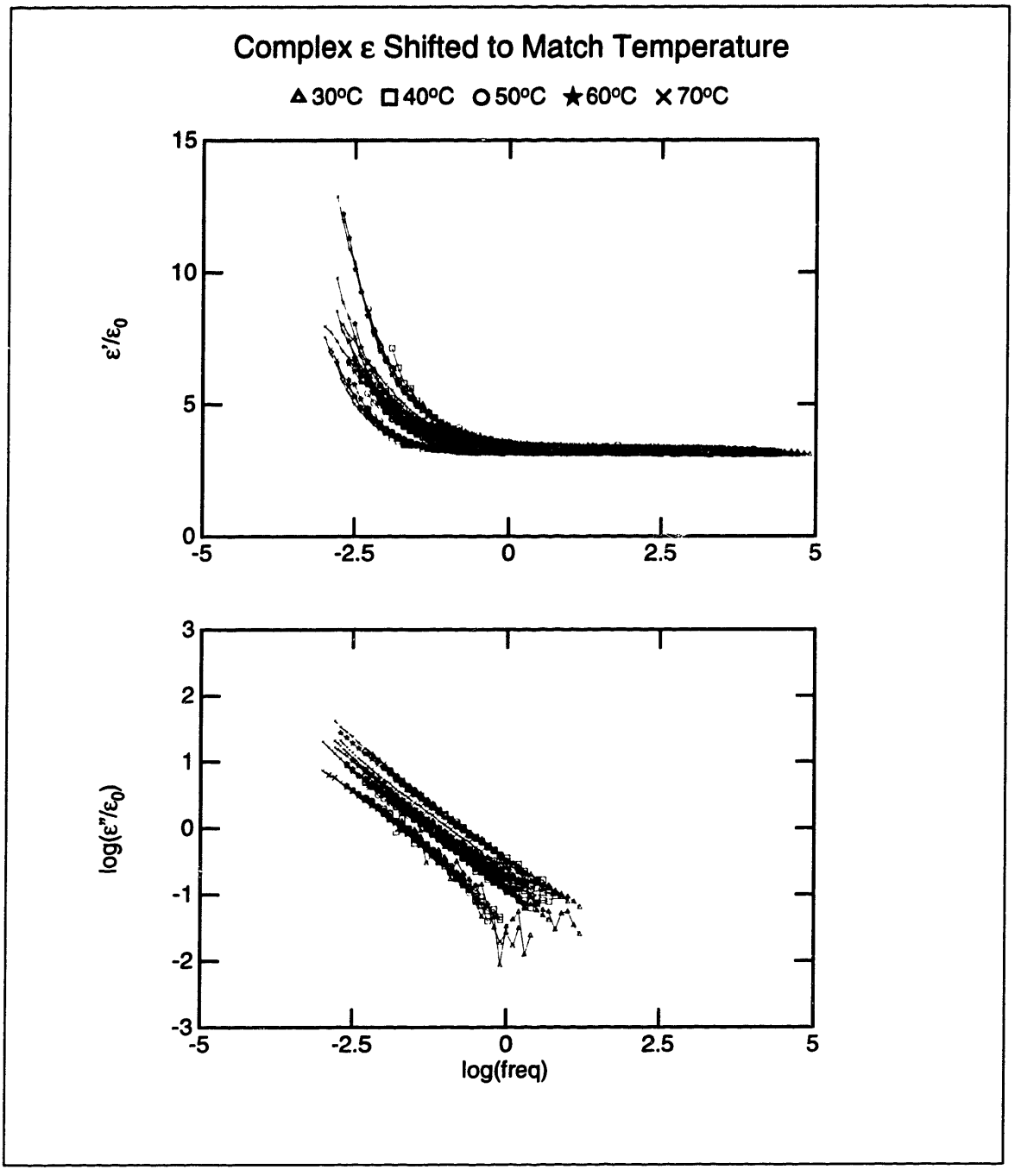

Figure 2-13: Temperature-shifted families of curves for the seven samples, each of the families being a universal spectrum for that sample

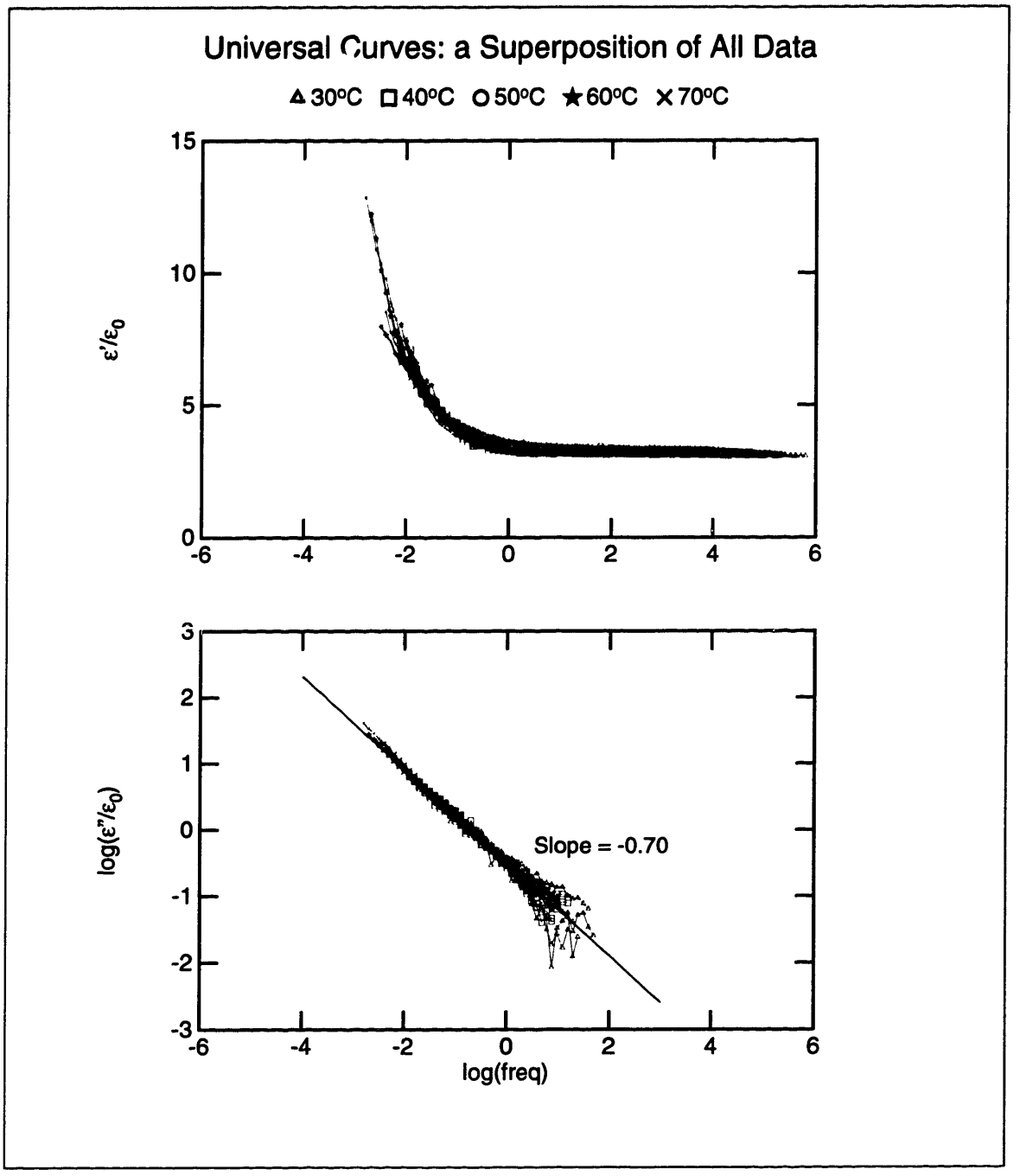

Figure 2-14: Master Universal Spectrum, containing data from 35 frequency scans, shifted with moisture and temperature

tion 2.1.3. Otherwise the thirty-five curves overlap quite closely. This universal mapping can now be used to estimate the moisture content of pressboard, if the dielectric spectrum is measured.

Table 2.3 lists the amounts by which each spectrum had to be shifted in order to form the master spectrum. These are logarithmic frequency shifts.

2.3.4 Correlation between the Frequency Shift and Temperature and Moisture

In order to know how the dielectric spectrum of pressboard changes with temperature and moisture content, we need to relate the logarithmic frequency shifts of Table 2.3 to the temperatures and moisture contents. Figure 2-13 implies that the effects of either of these factors are independent of each other. In order to test this hypothesis, we go on to perform some processing of the data in Table 2.3. What the independence mentioned above implies is that one can assign a quantity of shift to every temperature and to every moisture, and the shift that every spectrum is subjected to is the sum of the shifts due to these two factors. Symbolically, this may be represented as follows:

$$\log\left(\frac{\epsilon' - \epsilon_\infty}{\epsilon_0}\right) = \mathcal{F}'\{\log\omega - [f_T(T) + f_M(m)]\} \tag{2.36}$$

Name	Moist	30°C	40°C	50°C	60°C	70°C
MD	0.42%	-1.6	-1.3	-1.0	-0.7	-0.4
MF	0.83%	-1.8	-1.3	-1.0	-0.7	-0.3
ND	1.1%	-1.2	-0.9	-0.6	-0.3	0.1
MB	1.8%	-1.3	-1.0	-0.6	-0.3	0.2
MG	1.8%	-1.1	-0.7	-0.4	-0.2	0.1
MC	2.2%	-1.4	-0.9	-0.5	-0.2	0.2
MA	2.4%	-0.7	-0.4	**0.0**	0.4	0.5

Table 2.3: Relative logarithmic frequency shifts for data at different temperatures and moisture contents. Reference curves are at 50°C and 2.4% moisture (shown in bold).

	30°C	40°C	50°C	60°C	70°C
0.42%	-0.6	-0.3	**0.0**	0.3	0.6
0.83%	-0.8	-0.3	**0.0**	0.3	0.7
1.1%	-0.6	-0.3	**0.0**	0.3	0.5
1.8%	-0.7	-0.4	**0.0**	0.3	0.8
1.8%	-0.7	-0.3	**0.0**	0.2	0.5
2.2%	-0.9	-0.4	**0.0**	0.3	0.7
2.4%	**-0.7**	**-0.4**	**0.0**	**0.4**	**0.5**
Average	-0.71	-0.34	**0.0**	0.30	0.61

Table 2.4: In this table the spectra for all moisture contents have been shifted so that all of the 50°C curves overlap. In this way the effects due to moisture have been eliminated and one can calculate the average shift due to temperature.

	30°C	40°C	50°C	60°C	70°C	Average
0.42%	-0.9	-0.9	**-1.0**	-1.1	-0.9	-0.96
0.83%	-1.1	-0.9	**-1.0**	-1.1	-0.8	-0.98
1.1%	-0.5	-0.5	**-0.6**	-0.7	-0.4	-0.54
1.8%	-0.6	-0.6	**-0.6**	-0.7	-0.3	-0.56
1.8%	-0.4	-0.3	**-0.4**	-0.6	-0.4	-0.42
2.2%	-0.7	-0.3	**-0.5**	-0.6	-0.3	-0.48
2.4%	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

Table 2.5: In this table the spectra for all temperatures have been shifted so that all of the 2.4% curves overlap. In this way the effects due to temperature have been eliminated and one can calculate the average shift due to different moisture contents.

$$\log\left(\frac{\epsilon''}{\epsilon_0}\right) = \mathcal{F}''\{\log\omega - [f_T(T) + f_M(m)]\} \tag{2.37}$$

where $f_T(T)$ depends only on the absolute temperature and $f_M(m)$ depends only on moisture. These formulas also incorporate the requirement that both components of ϵ^* shift by the same amount (see Appendix A).

The strategy applied to test the validity of the assumption that the shifts due to temperature and moisture are independent is to take every row in Table 2.3 and add to (or subtract from) every number in it the same amount in a way that would make the shift at 50°C be zero. This operation results in the numbers shown in Table 2.4. We have normalized the data in such a way that the effect of moisture has been

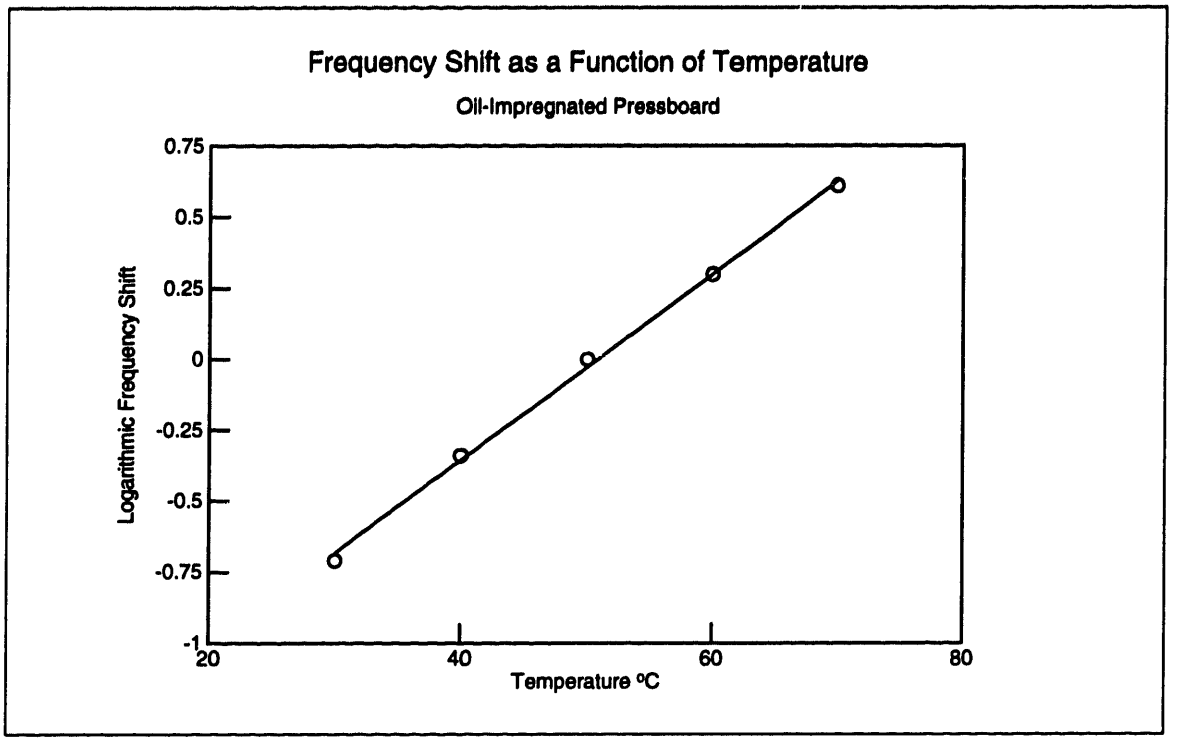

Figure 2-15: Logarithmic frequency shift as a function of temperature

eliminated. If the two effects are truly independent of each other, the numbers in every column of Table 2.4 would be approximately the same, which would represent the frequency shift due to temperature alone. We can see in Table 2.4 that this is apparently true. We can now take the average of the numbers in every column to be the logarithmic shift due to temperature, as plotted in Figures 2-15 and 2-16.

We can similarly eliminate the effects of temperature by making the entire row in Table 2.3 for MA be all zeros, by adding or subtracting the appropriate amount from each number in the same column. The results of this operation are shown in Table 2.5. From this table we can now calculate the average logarithmic shift due to moisture, as listed in Table 2.5 and plotted in Figure 2-17. In conclusion we can say that indeed the two factors independently shift the dielectric spectrum of pressboard.

One can see in Figure 2-15 that the relationship between temperature and logarithmic frequency shift is approximately linear. However, this is of dubious significance in light of the fact that only a small interval of temperatures are spanned on an absolute temperature scale; 70°C is only 13.2% higher than 30°C. This means

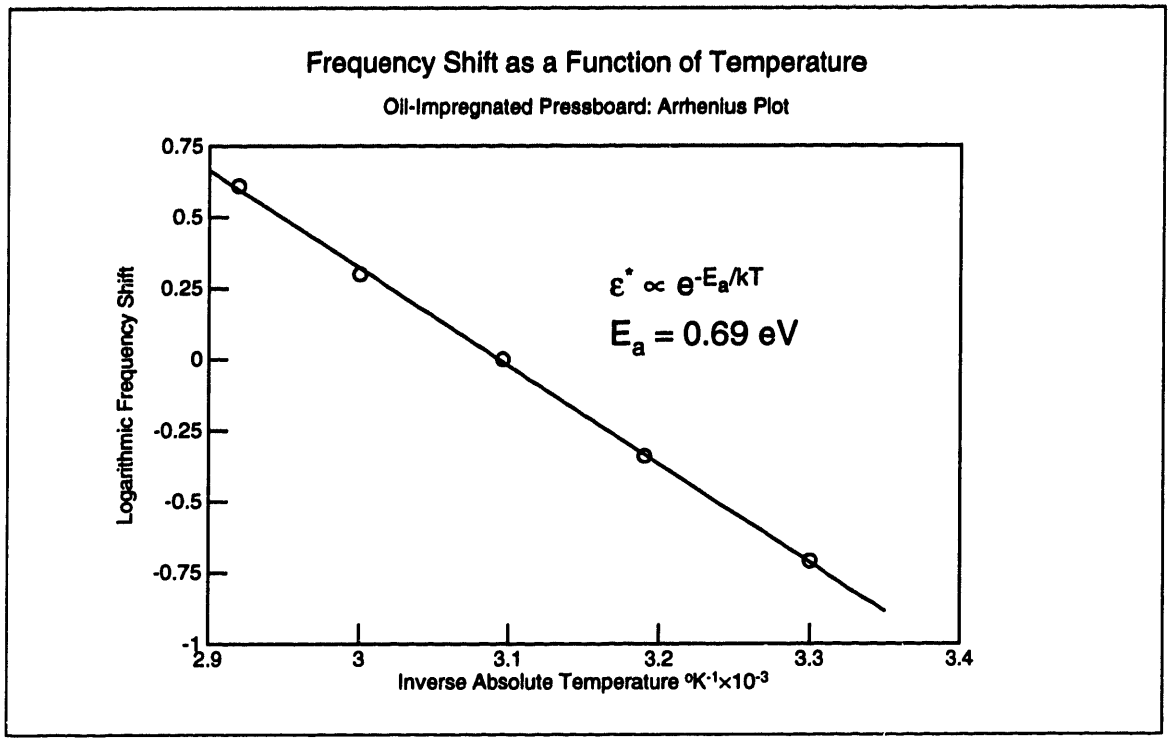

Figure 2-16: Logarithmic frequency shift as a function of temperature: Arrhenius plot

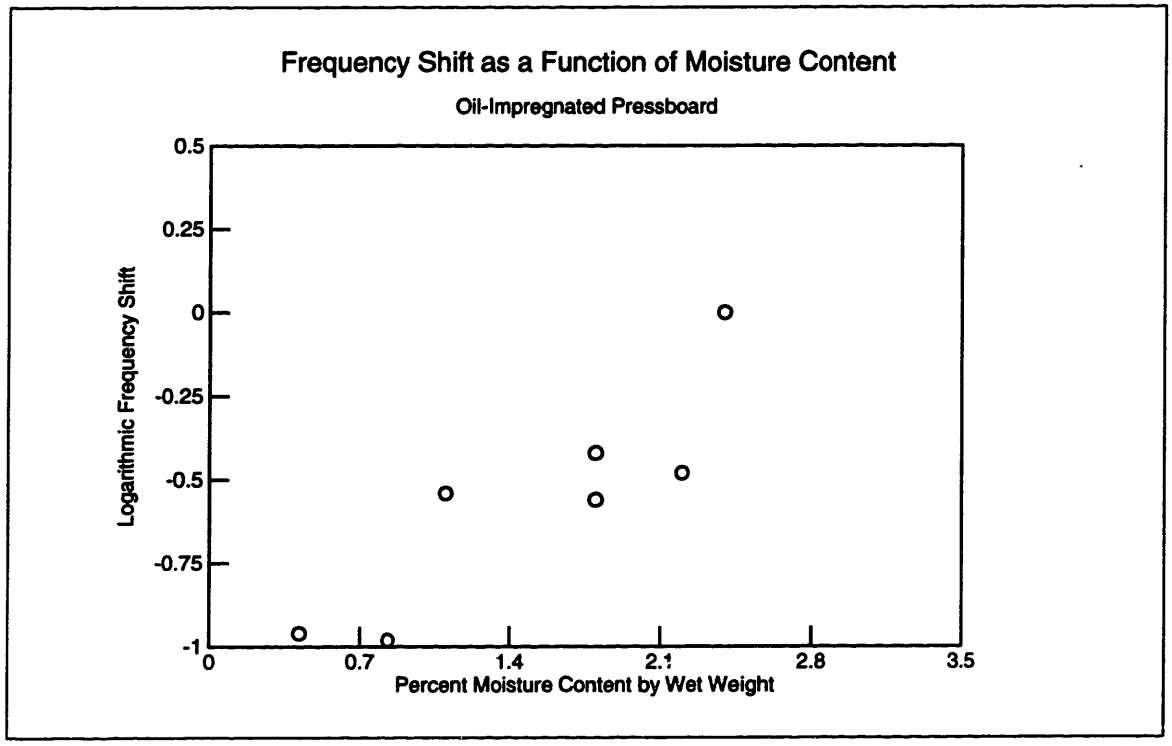

Figure 2-17: Logarithmic frequency shift as a function of moisture

that almost any functional dependence may appear linear over a small interval like that. For example, if we assume the Arrhenius dependence $\epsilon^* \propto e^{-E_a/kT}$, where k is Boltzmann's constant, then the plot of the logarithmic shift versus the inverse of the absolute temperature should be a straight line. Figure 2-16 indeed shows that the fit to this functional form is as good as the one in Figure 2-15. The slope in Figure 2-16 corresponds to an activation energy of $E_a = 0.69$ eV. Either of these plots may be used to obtain the frequency shift associated with a value for the temperature.

Figure 2-17 shows how the logarithmic frequency shift depends on the moisture content. Since the moisture measurements have a relatively large margin of error (see Appendix B), seven data points are certainly insufficient to establish a functional dependence. Figure 2-17 may be used as an empirical relation, but many more data points would be necessary if this curve is to be reliably evaluated.

2.4 Algorithm for Using the Universal Spectrum

Suppose that we perform a dielectrometry measurement on a sample of pressboard at a known temperature. How can we use the results presented in Section 2.3 to find its moisture content?

First of all, we need to determine what kind of a frequency shift would map this spectrum onto the reference spectrum MA at 50°C. This is most easily accomplished with the help of the program **fith.c**, listed in Section G.4. If the dielectrometry is performed only at a single frequency, then Figure 2-14 can be used to determine at what frequency the corresponding value of ϵ^* is achieved, and the frequency shift will be equal to the difference between these two frequencies.

The next step would be to find the frequency shift associated with the temperature of the sample. This may be done graphically in Figures 2-15 or 2-16. The shift due to temperature is then subtracted from the total shift and we are left with the shift due to moisture. Finally, Figure 2-17 is used to see what moisture content would correspond to this frequency shift.

Note that the logarithmic shifts due to moisture and temperature can be both

positive or negative numbers, depending on the choice of a reference spectrum. Attention should be paid to the signs of these quantities when applying the algorithm described above.

Chapter 3

The Flexible Three-Wavelength Interdigital Sensor

3.1 Structure

The flexible three-wavelength sensor uses the ideas presented in Section 1.4. Its structure is shown in Figure 3-1. It consists of three sets of interdigitated electrodes, deposited on a common flexible Kapton (a polyimide) substrate. Every set of electrodes contains ten wavelengths. The area of the active surface is about $2'' \times 2''$. In a way similar to the parallel-plate sensor (see Section 2.1), the sensing electrodes of every wavelength are shielded by guard electrodes, driven by the buffer stage in the interface box (see Appendix E), and the guard electrodes are shielded by ground electrodes. All of the electrodes connect to the interface box via the flexible leads.

On the bottom surface of the substrate a copper ground plane is deposited, which is electrically connected to the ground electrodes. The entire sensor is coated with Parylene, a hydrophobic polymer, which serves to protect the sensor from contamination. Table 3.1 lists the physical parameters of the three-wavelength sensor [2, sec. 6.3].

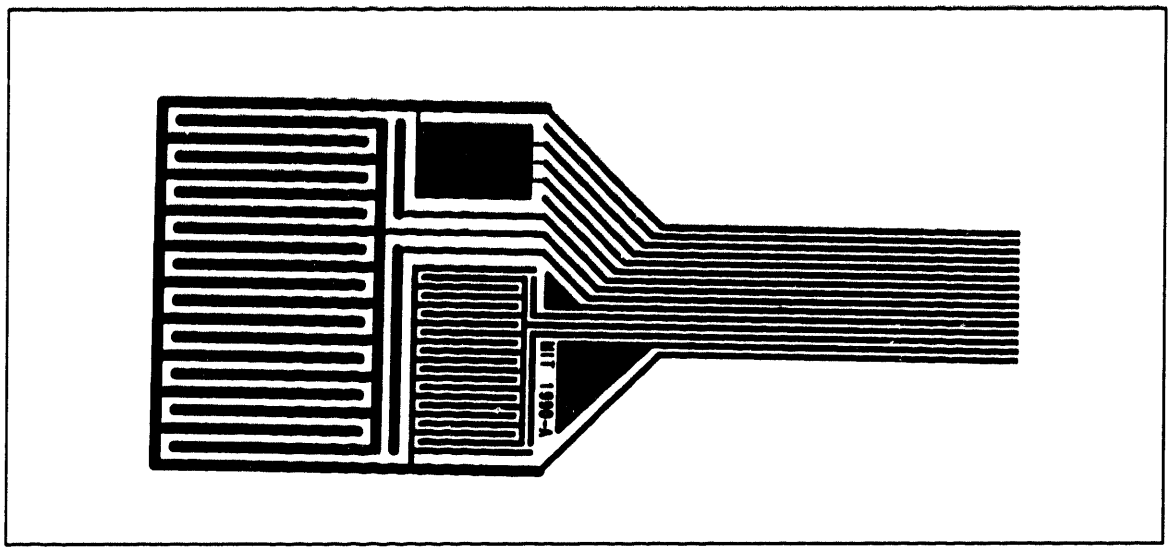

Figure 3-1: Structure of the three-wavelength interdigitated sensor [2]

Parameter	Symbol	Values		
Spatial wavelength	λ	1.0 mm	2.5 mm	5.0 mm
Interelectrode spacing	a	0.24 λ	0.24 λ	0.24 λ
Electrode meander length	M_L	0.15 m	0.15 m	0.30 m
Substrate thickness	h	127 μm	127 μm	127 μm
Substrate permittivity	ϵ_{OX}	3.0 ϵ_0	3.0 ϵ_0	3.0 ϵ_0
Parylene layer thickness	d_{PX}	5.0 μm	5.0 μm	5.0 μm
Permittivity of Parylene	ϵ_{PX}	3.05 ϵ_0	3.05 ϵ_0	3.05 ϵ_0

Table 3.1: Values of parameters describing the three-wavelength sensor [2]

3.2 Manufacturing

The major issue in the manufacturing of the three-wavelength sensor is maintaining a clean electrode surface. Since the materials being tested are highly insulating, the sensor is extremely sensitive to surface conductivity at the plane of the electrodes.

The actual process involves three stages: At the first stage the electrode pattern is formed on the Kapton substrate by depositing a conducting layer of copper and then using a mask to selectively etch the pattern[1]. At the second stage vapor deposition is used to form the ground plane on the other side of the substrate. Finally, the sensor

[1]MIT Part DOFLEX, Rev. 120390, Tech-Etch, Inc., 45 Aldrin Road, Plymouth, MA 02360, (617) 747-0300.

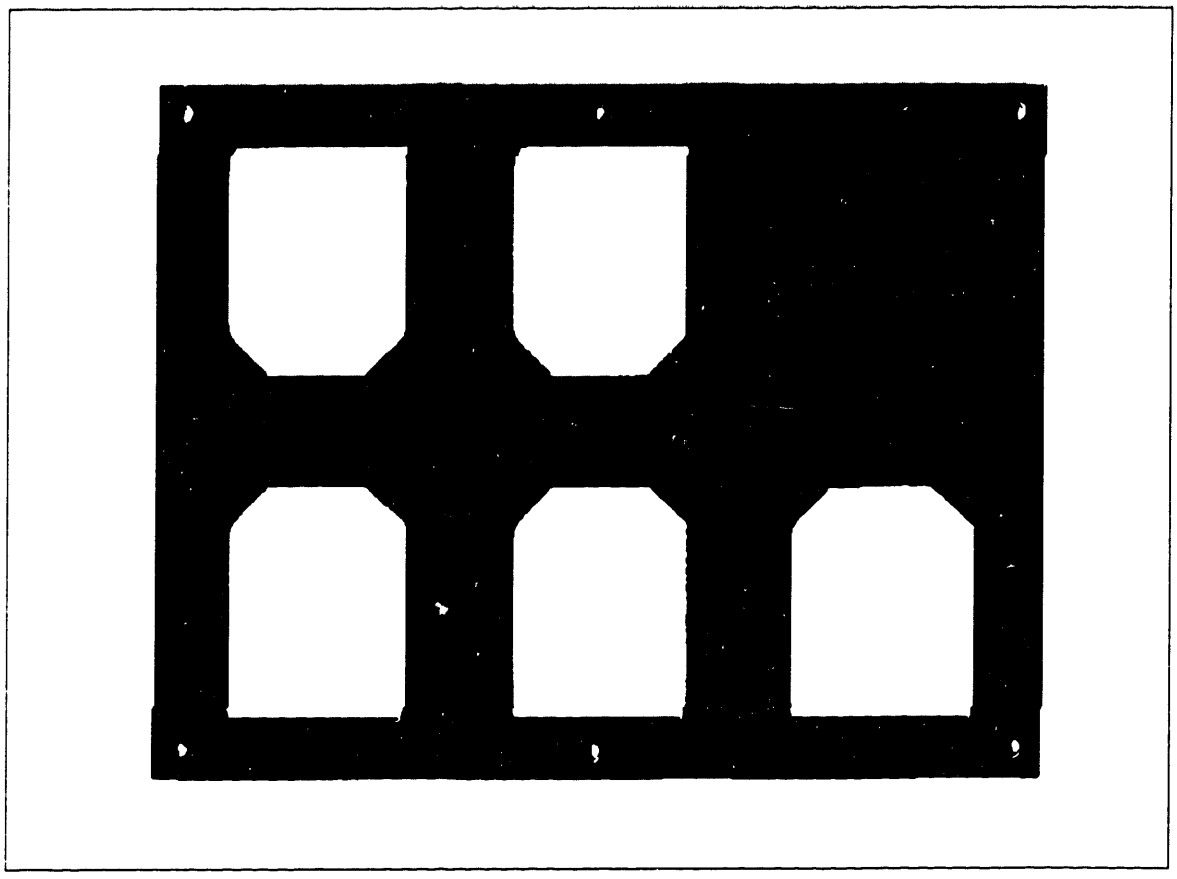

Figure 3-2: Mask used for the copper back plane deposition.

is coated for protection with a layer of Parylene. See [2, sec. 6.2.2] for a more detailed description of the manufacturing process than presented in this section.

The electrode pattern was formed by selectively etching copper from a copper/Kapton composite. To prevent oxidation the electrodes were gold-plated. The copper ground plane was deposited at the back of the sensor in an electron-beam vapor deposition chamber[2], where a physical mask served to expose the surface where copper was to be deposited. This mask allows for the processing of five sensors simultaneously. It is shown in Figure 3-2. The copper layer thickness is about 1 μm.

At this point the sensors were ready for the cleaning process, which was the most critical stage in ensuring that the sensors be operational. There are two stages to the cleaning process: chemical cleaning, which involves rinsing with solvents, and plasma

[2]Microelectronics Technologies Laboratory, Center for Materials Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139

etching. Plasma etching is accomplished by ionizing oxygen in a vacuum chamber, which then reacts with the Kapton and with any contaminants on the surface. All plasma etching stages described in this section were done at 300 W for a length of two minutes.

The criterion for determining whether a sensor is "clean" is that in air its dielectric response should have a constant gain and zero phase for the full range of frequencies between 0.005 Hz and 10 kHz. This would mean that the sensor's impedance was purely capacitive. Any phase angle visible at the lower range of frequencies would imply conduction between the electrodes due to contaminants.

The procedure we followed at first involved rinsing with acetone and methanol and plasma etching. Before we went on to the next stage, we tested the three sensors in air and discovered that the cleaning had been quite ineffective, as can be seen in Figure 3-3, which shows the gain and phase of the response of this sensor in air. The conduction is very noticeable at frequencies below 1 Hz. The other two sensors showed similar results. Although this short cleaning procedure had been sufficient in the past [2], it did not produce the desired results with this set of sensors, apparently because of a higher initial level of contaminants.

We then changed the chemical cleaning protocol to the following:

1. Rinse with trichloroethylene ($C_2HC\ell_3$)
2. Rinse with acetone (CH_3COCH_3)
3. Rinse with methanol (CH_3OH)
4. Rinse with deionized water (H_2O)

with the important requirement that the sensor not be let dry between the different rinsing stages, in order that every subsequent solvent dissolve any residue left by the preceding one. We also added a heating stage, during which the sensor is kept at a high temperature in air (50–70°C) for a few hours, to evaporate any water left on the surface after the last rinsing stage.

The first sensor was chemically cleaned by the four-step procedure and then heated at 50°C for 18 hours. The sensor appeared perfectly clean (flat dielectric response with frequency) even before the plasma etching, as shown in Figure 3-4. In order to investigate whether all of these stages are needed, we eliminated the four solvent stages for the second sensor and only subjected it to heating: 20 hours at 60°C. This method was not satisfactory because the resulting dielectric spectrum was like that in Figure 3-3. Then we applied the solvent cleaning as described above, but did not subject the sensor to heating. This was equally ineffective in yielding a clean sensor. Finally, we tested to see whether the entire 20-hour period of heating is really necessary, by treating the third sensor with solvents and heat at 70°C for one hour. This procedure produced a clean sensor and further heating had no appreciable effect.

We therefore established the following protocol for the chemical cleaning of the interdigital flexible sensors:

1. Rinse with trichloroethylene ($C_2HC\ell_3$)
2. Rinse with acetone (CH_3COCH_3)
3. Rinse with methanol (CH_3OH)
4. Rinse with deionized water (H_2O)
5. Heat in oven in air at 70°C for 1 hour

The Parylene coating process involved keeping the sensors under vacuum for 12 hours, which acted to remove moisture and any other volatile contaminants from the bulk of the Kapton substrate. Without exposure to ambient conditions, the sensors were coated with 5 μm of Parylene (Poly(para-xylylene)). The coating process resulted in a pin-point free water-resistant protective layer.

A final test of the ready sensors demonstrated that they were still clean because their dielectric spectra were flat when tested in air.

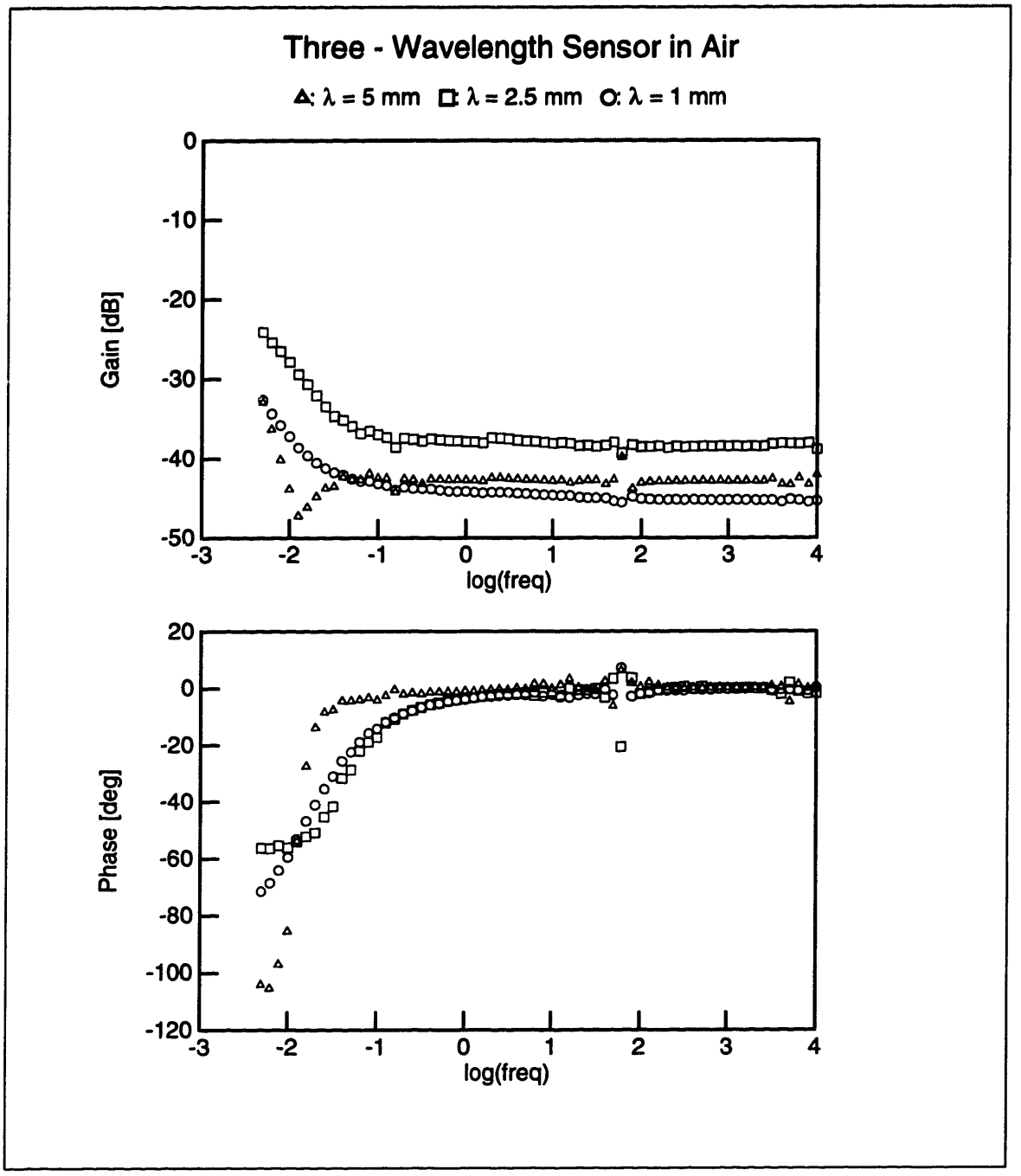

Figure 3-3: Response of a three-wavelength sensor in air before chemical cleaning and before Parylene coating. It shows non-zero phase and increasing gain at low frequencies, which indicates contamination.

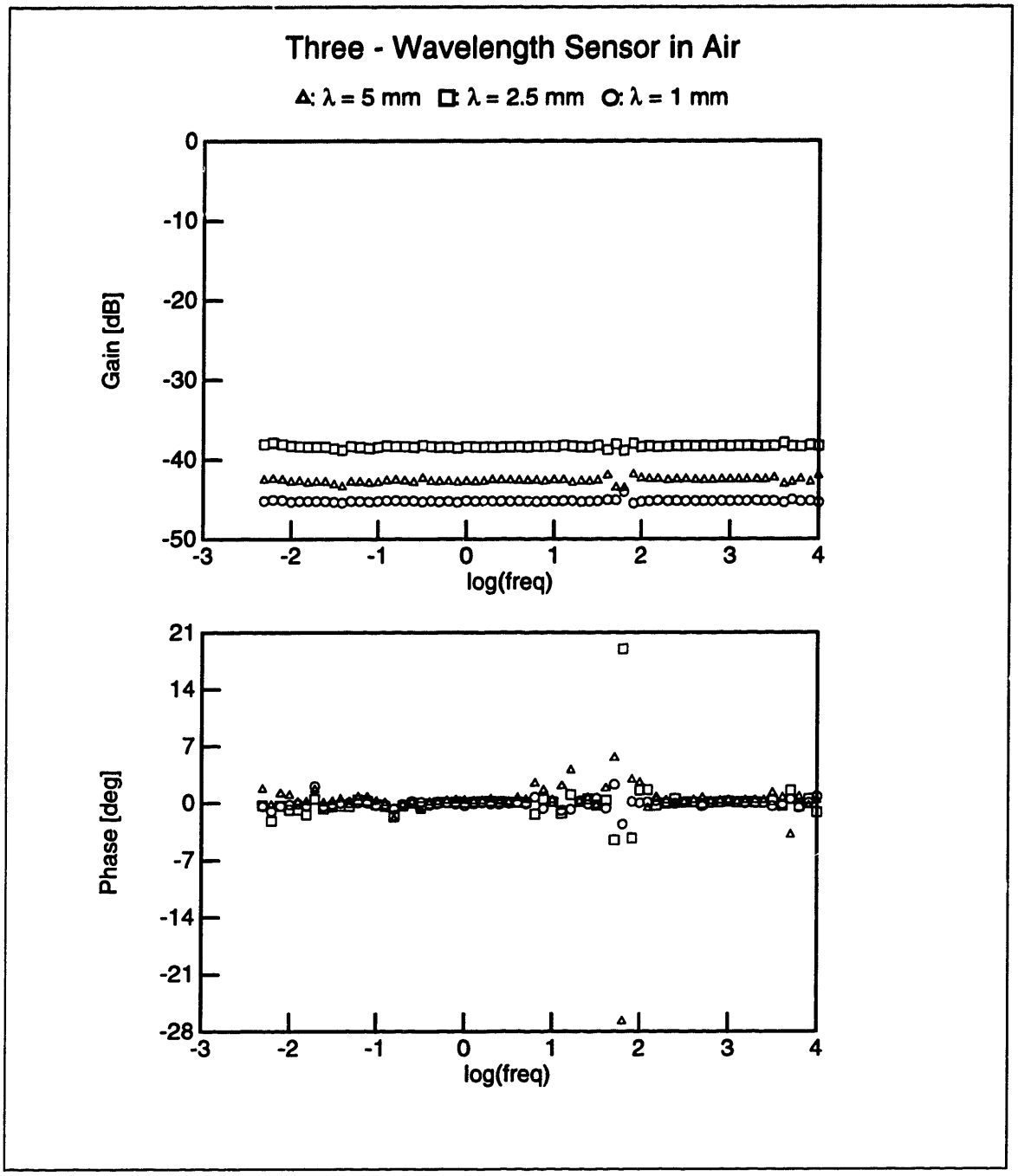

Figure 3-4: Response of a three-wavelength sensor in air before Parylene coating and after recommended chemical cleaning procedure and heating. It shows zero phase and constant gain over the entire frequency range, indicating a clean sensor.

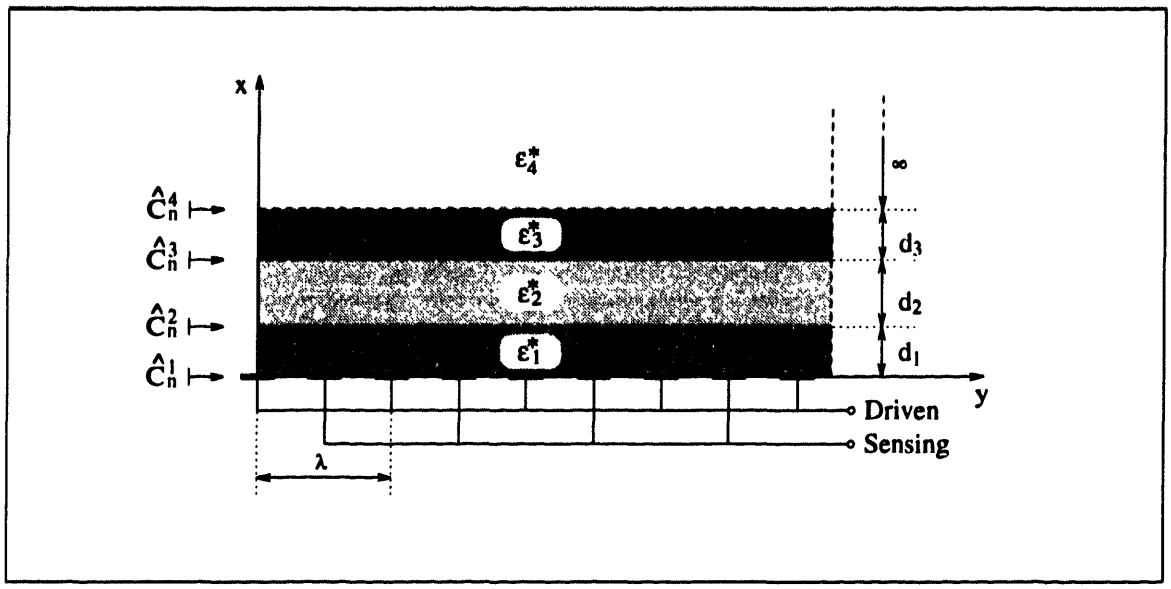

Figure 3-5: Interdigitated electrode structure with a number of homogeneous layers above it

3.3 Mathematical Model

Finding the admittance of the parallel-plate sensor as a function of the material properties and the geometry of the test cell is trivial, since the electric fields are highly uniform and one-dimensional (see Section 2.1).

The task of finding the complex impedance of an interdigitated structure as a function of the properties of all materials and geometric considerations is much more complicated, because the fields are two-dimensional and the potential distribution on the surface between the electrodes is not known and must be calculated from conservation of charge. In fact, these complications make it impossible to express the impedance of the sensor in closed form, and in general numerical methods are needed to calculate the impedance. In this section we present a summary of the procedures applied in obtaining a mathematical model for the interdigitated sensor. For a more detailed discussion see [3] [7].

Let us suppose that we have the interdigitated electrode structure shown in Figure 3-5. For the purposes of the discussion in this section, we idealize the structure by assuming no z-dependence of the electric fields and by assuming electrodes of negligible thickness. We are also assuming that there is no surface conductivity at the

interfaces between different material layers, although such effects are easy to incorporate in the model [3].

In this discussion we use the following convention: All quantities are complex since only steady-state AC excitation is assumed. Such complex amplitudes that are functions of space are denoted by a 'tilde' ($\tilde{}$). The time dependence of the corresponding physical quantity is obtained from the following formula:

$$F(x,y,t) = \Re\left\{\tilde{F}(x,y)e^{j\omega t}\right\} \tag{3.1}$$

where ω is the steady-state radian frequency. If a quantity's spatial y-dependence is also sinusoidal, then it can be represented by a complex phasor denoted by a 'hat' ($\hat{}$):

$$\tilde{F}(x,y) = \hat{F}(x)e^{-jky} \tag{3.2}$$

$$F(x,y,t) = \Re\left\{\hat{F}(x)e^{j(\omega t-ky)}\right\} \tag{3.3}$$

where k is the wave number, related to the wavelength λ as

$$k = \frac{2\pi}{\lambda} \tag{3.4}$$

Every interdigitated section of the three-wavelength sensor has three electrical terminals: driven electrode, sensing electrode, and ground plane. The guard electrode is always at the same potential as the sensing electrode, so that any coupling between them is effectively eliminated. Any admittance between the guard electrode and ground or the driven electrode has no influence on the measurement. We may therefore leave the guard electrode out of the circuit model. Our goal is to be able to calculate the admittances between these three terminals from the parameters of the layer structure.

The circuit model of the structure is shown in Figure 3-6. The admittance Y_{11} of the driven electrode to ground is the same as the admittance of the sensing elec-

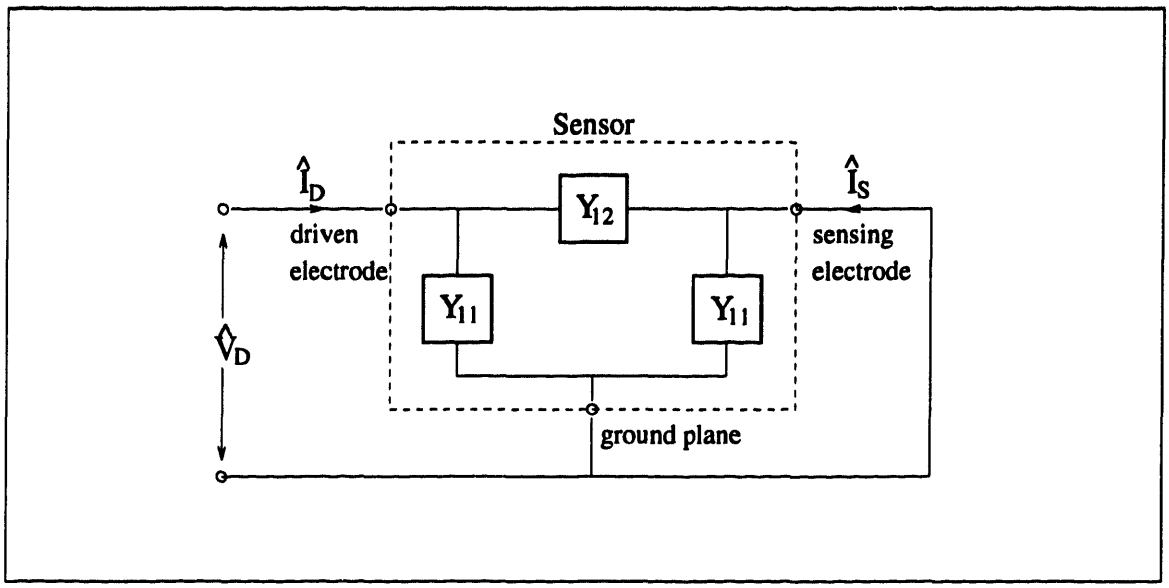

Figure 3-6: Lumped circuit model for the interdigitated sensor structure within dashed box and shown with grounded sensing electrode.

trode to ground because the two electrodes have an identical geometry. Y_{12} represents the coupling between the driven and the sensing electrodes. We may calculate the admittances by applying test voltages at the terminals and calculating the resulting terminal currents. The simplest form of this test drive is to ground the sensing electrode and apply a voltage $\hat{V}_D$ to the driven electrode. Then the unknown admittances can be calculated in the following way:

$$Y_{12} = -\frac{\hat{I}_S}{\hat{V}_D} \tag{3.5}$$

$$Y_{11} = \frac{\hat{I}_D + \hat{I}_S}{\hat{V}_D} \tag{3.6}$$

In order to calculate $\hat{I}_D$, we need to know the normal component of the total (displacement plus conduction) current density $j\omega\epsilon^* \tilde{E}_x(y)$, which is integrated over the area of the driven electrode to give the total terminal current. It is therefore necessary to solve for the electric field distribution.

The entire interdigitated structure is periodic in the y-direction with a wavelength λ. This means that for every quantity that depends on y we may use Fourier series

expansions to write:

$$\tilde{\Phi}(x,y) = \sum_{n=-\infty}^{\infty} \hat{\Phi}_n(x) e^{-jk_n y} \tag{3.7}$$

$$\tilde{E}_x(x,y) = \sum_{n=-\infty}^{\infty} \hat{E}_{xn}(x) e^{-jk_n y} \tag{3.8}$$

where n is the Fourier mode number and

$$k_n = \frac{2\pi n}{\lambda} \tag{3.9}$$

It is convenient to define the *complex surface capacitance density* $\hat{C}_n$, which relates $\epsilon^* \hat{E}_{xn}$ at a planar surface x = constant to the potential $\hat{\Phi}_n$ at that surface for every Fourier mode n in the following way:

$$\hat{C}_n = \frac{\epsilon^* \hat{E}_{xn}}{\hat{\Phi}_n} \tag{3.10}$$

Knowing $\hat{C}_n$ at the electrode surface will let us calculate the terminal currents from the potential distribution at that surface.

We have assumed that every layer of material in Figure 3-5 is uniform, i.e. the complex permittivity $\epsilon^* = \epsilon - j\sigma/\omega$ is independent of the spatial coordinates. This means that Laplace's equation

$$\nabla^2 \Phi = 0 \tag{3.11}$$

is satisfied everywhere in space except at the interfaces between the layers. At these interfaces, however, the boundary conditions require continuity of $\hat{\Phi}$ (tangential component of $\vec{E}$ is continuous) and $\epsilon^* \hat{E}_x$ (normal component of conduction plus displacement current density is continuous). This means that at these interfaces $\hat{C}$ may be uniquely defined. Let $\hat{C}_n^m$ be the complex surface capacitance density at the interface between the mth layer and the one below it (see Figure 3-5), i.e. the surface at $x = \sum_{i=1}^{m-1} d_i$, with n referring to the Fourier mode. If we can express $\hat{C}_n^m$ in terms of $\hat{C}_n^{m+1}$, d_m, and ϵ_m^*, then we could apply this relationship recursively, beginning at the

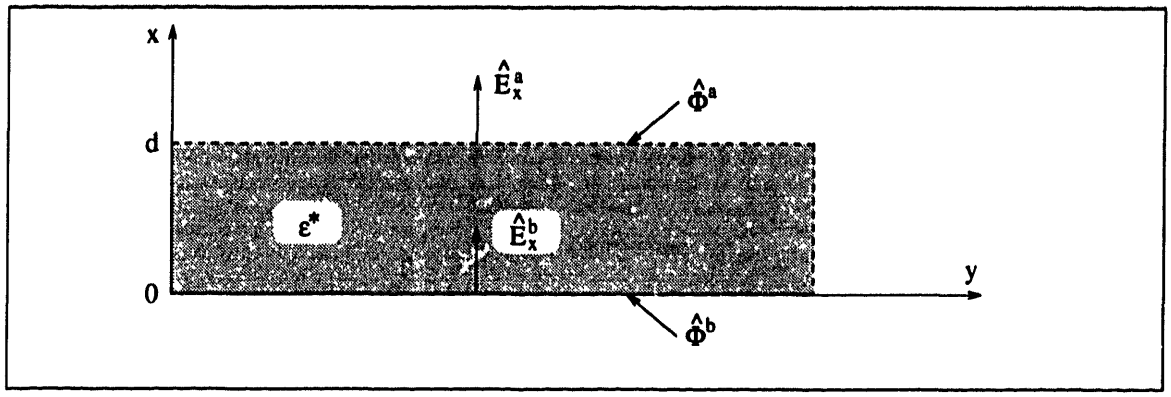

Figure 3-7: A representative layer of homogeneous material

topmost layer N, to obtain $\hat{C}_n^N$, then $\hat{C}_n^{N-1}$, etc. and ultimately find $\hat{C}_n^1$, the complex surface capacitance density at the electrode surface.

Consider the slab of uniform material in Figure 3-7. We would like to find $\hat{C}_n^b = \epsilon^* \hat{E}_x^b / \Phi^b$ as a function of $\hat{C}_n^a = \epsilon^* \hat{E}_x^a / \Phi^a$, ϵ^*, k_n, and d. Since equation 3.11 is satisfied, and since the potential is periodic in the y-direction with a wave number of k_n, the x-dependence of $\hat{\Phi}$ must be exponential. We therefore guess the following form:

$$\hat{\Phi}(x) = \hat{A} \sinh k_n x + \hat{B} \cosh k_n x \tag{3.12}$$

We have the boundary conditions

$$\hat{\Phi}(x = 0) = \hat{\Phi}^b \tag{3.13}$$

$$\hat{\Phi}(x = d) = \hat{\Phi}^a \tag{3.14}$$

which leads by inspection to the final solution for $\hat{\Phi}$:

$$\hat{\Phi}(x) = \frac{\hat{\Phi}^a \sinh k_n x - \hat{\Phi}^b \sinh k_n (x - d)}{\sinh k_n d} \tag{3.15}$$

The x-directed electric field can be obtained from equation 3.15 by differentiation as follows:

$$\hat{E}_x = -\frac{d\hat{\Phi}}{dx} = k_n \left[\frac{-\hat{\Phi}^a \cosh k_n x + \hat{\Phi}^b \cosh k_n (x - d)}{\sinh k_n d} \right] \tag{3.16}$$

from where we obtain

$$\hat{E}_x^a = \hat{E}_x(x=d) = -k_n\hat{\Phi}^a \coth k_n d + k_n\hat{\Phi}^b \frac{1}{\sinh k_n d} \tag{3.17}$$

$$\hat{E}_x^b = \hat{E}_x(x=0) = -k_n\hat{\Phi}^a \frac{1}{\sinh k_n d} + k_n\hat{\Phi}^b \coth k_n d \tag{3.18}$$

$$\hat{C}^a = \epsilon^* k_n \left(-\coth k_n d + \frac{\hat{\Phi}^b}{\hat{\Phi}^a \sinh k_n d} \right) \tag{3.19}$$

$$\hat{C}^b = \epsilon^* k_n \left(-\frac{\hat{\Phi}^b}{\hat{\Phi}^a \sinh k_n d} + \coth k_n d \right) \tag{3.20}$$

From equation 3.19 we obtain

$$\frac{\hat{\Phi}^b}{\hat{\Phi}^a} = \left(\frac{\hat{C}^a}{\epsilon^* k_n} + \coth k_n d \right) \sinh k_n d \tag{3.21}$$

which we can then substitute into equation 3.20 to yield:

$$\begin{aligned}
\hat{C}^b &= \epsilon^* k_n \left[-\frac{1}{\left(\frac{\hat{C}^a}{\epsilon^* k_n} + \coth k_n d \right) \sinh^2 k_n d} + \coth k_n d \right] \\
&= \epsilon^* k_n \left(\coth k_n d - \frac{\frac{\epsilon^* k_n}{\sinh^2 k_n d}}{\hat{C}^a + \epsilon^* k_n \coth k_n d} \right) \\
&= \epsilon^* k_n \left(\frac{\hat{C}^a \coth k_n d + \epsilon^* k_n \coth^2 k_n d - \frac{\epsilon^* k_n}{\sinh^2 k_n d}}{\hat{C}^a + \epsilon^* k_n \coth k_n d} \right) \\
&= \epsilon^* k_n \left(\frac{\hat{C}^a \coth k_n d + \epsilon^* k_n}{\hat{C}^a + \epsilon^* k_n \coth k_n d} \right) \\
&= \epsilon^* k_n \left(\frac{\hat{C}^a \cosh k_n d + \epsilon^* k_n \sinh k_n d}{\hat{C}^a \sinh k_n d + \epsilon^* k_n \cosh k_n d} \right)
\end{aligned} \tag{3.22}$$

Let us test the validity of equation 3.22 in the limits $d = 0$ and $d = \infty$. For a layer of zero width this equation reduces to $\hat{C}^b = \hat{C}^a$, as required. For $d \to \infty$ both

the hyperbolic sine and cosine approach the exponential function, i.e.

$$\lim_{d\to\infty}\left(\frac{\widehat{C}^a\cosh k_n d+\epsilon^* k_n\sinh k_n d}{\widehat{C}^a\sinh k_n d+\epsilon^* k_n\cosh k_n d}\right)=1 \tag{3.23}$$

which reduces equation 3.22 to $\widehat{C}^b=\epsilon^* k_n$. This is a useful result, as it directly applies to the semi-infinite topmost layer in Figure 3-5. If its index number is N, then

$$\widehat{C}_n^N=\epsilon_N^* k_n \tag{3.24}$$

We now have the means of calculating $\widehat{C}_n^1$ from equations 3.22 and 3.24 by recursively descending down the layer structure. If on the bottom side of the electrode plane we had a similar set of layers, we would obtain a value for the surface capacitance density from that side too.

Instead of structure of layers similar to the one in Figure 3-5, the bottom side of the three-wavelength sensor has a single substrate layer of thickness h and permittivity ϵ_{OX}[3], which is actually purely real because the Kapton substrate's conductivity is negligible, as illustrated in the next section. On the other side of the substrate the ground plane is deposited. We cannot use the equations developed so far to obtain the surface capacitance density due to the bottom side of the electrodes, because the potential at the ground plane is forced to zero. We may, however, use equation 3.19 with $\widehat{\Phi}^b=0$ to obtain

$$\widehat{C}_n^{-1}=-\epsilon_{OX}k_n\coth k_n h \tag{3.25}$$

where the negative superscript indicates layers below the surface.

Since the surface capacitance density is known, it could be integrated over the areas of the driven and the sensing electrodes to obtain the currents $\widehat{I}_D$ and $\widehat{I}_S$ if the potential is known at the electrode plane. While the potential is indeed known along the electrodes, where it is constrained by them to be $\widehat{V}_D$ (driven electrode) or zero (grounded sensing electrode), in the space between the electrodes it is not known

[3]The name ϵ_{OX} is kept for historical reasons from a time when the substrate was manufactured from an oxide material. Kapton is a polyimide, not an oxide.

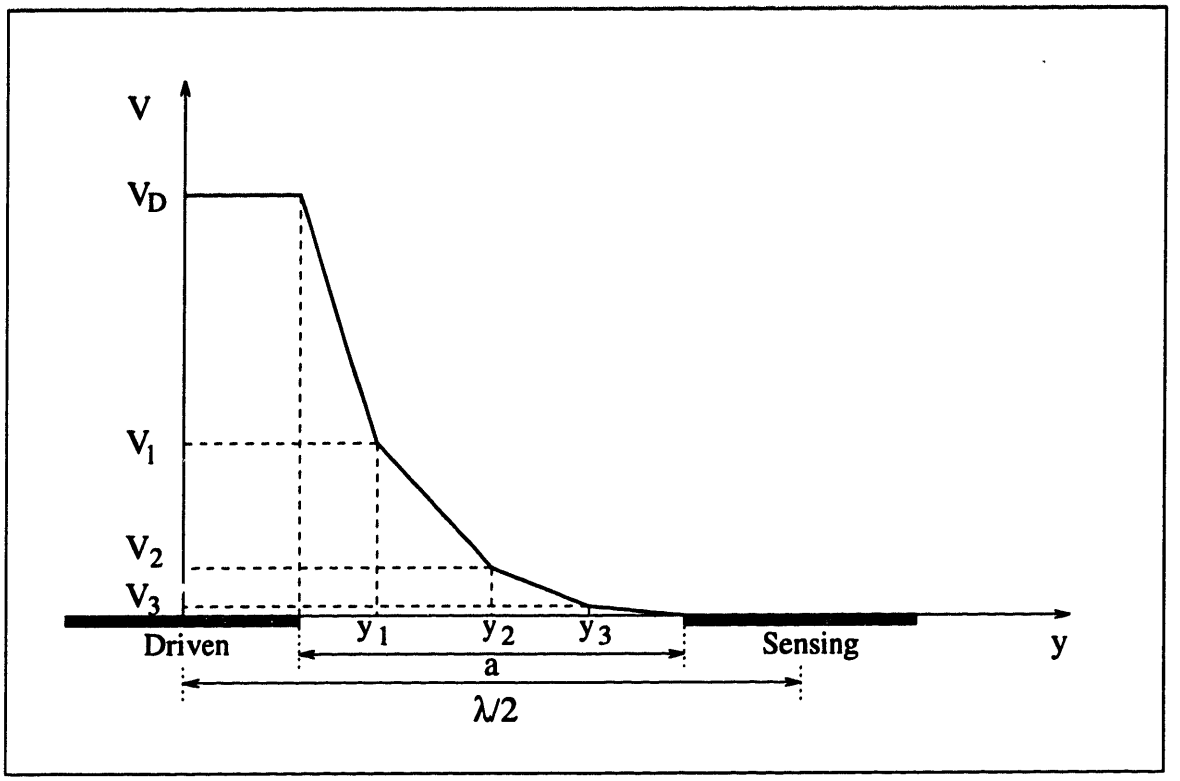

Figure 3-8: Piecewise-smooth collocation-point approximation to the potential between the electrodes of an interdigitated structure. Three collocation points at y_1, y_2, and y_3 are shown

and must be determined by a different boundary condition, namely conservation of charge.

The potential between the electrodes is approximated by a piecewise-linear function, which divides the space between the electrodes into $k+1$ intervals, delimited by k *collocation points*, as shown in Figure 3-8. In every interval the potential is assumed to vary linearly between the potentials at the two end points. The potential distribution is thus fully determined by the potential at the k collocation points. Now that we have a form for $\tilde{\Phi}(y)$, we can use the Fourier integral to obtain an expression for $\hat{\Phi}_n$, which is an algebraically linear function of the unknown potentials at the collocation points. In order to find these potentials $\hat{V}_j$, we need a set of k equations, which can be obtained by applying conservation of charge to k intervals centered around the collocation points. Reference [3] presents this process in detail, carrying out all integrations, etc. What is important to us is that this numerical process yields

the potential distribution at the electrode surface and ultimately makes it possible to find Y_{11} and Y_{12}.

Finally, if the admittances in Figure 3-6 are known and if the sensor is loaded by an interface box of input admittance Y_L (see Appendix E), the magnitude and phase of the voltage ratio are given by

$$\frac{\hat{V}_S}{\hat{V}_D} = \frac{Y_{12}}{Y_{12} + Y_{11} + Y_L} \tag{3.26}$$

$$M = \left|\frac{\hat{V}_S}{\hat{V}_D}\right| = \left|\frac{Y_{12}}{Y_{12} + Y_{11} + Y_L}\right| \tag{3.27}$$

$$\varphi = \angle\left(\frac{\hat{V}_S}{\hat{V}_D}\right) = \angle\left(\frac{Y_{12}}{Y_{12} + Y_{11} + Y_L}\right) \tag{3.28}$$

The subsidiary parameter estimation routines, listed in Section H.5, implement the numerical calculations presented in this section. The function **gp()**, defined in **gp.c** gives the gain and phase of the voltage ratio of an interdigitated sensor when it is supplied with the parameters of the sensor and with the properties of the layers above it. The code used in its implementation is an almost direct translation from PASCAL of the code written by Dr. M. Zaretsky [3].

There are also some parasitic effects, due to the finite thickness of the electrodes, which result in a parasitic admittance in parallel with Y_{12}. See [2, sec. 6.2.3] for a discussion of these effects.

In a number of our measurements an aluminum plate was used to squeeze the materials against the surface of the interdigitated sensor. In those cases the topmost layer had to be approximated as infinitely conducting. The presence of a conductor at the top does not introduce anything new to the analysis, provided it is left floating, as the model assumes. If the conductor is grounded, the admittance network of Figure 3-6 would be altered and this change must be included in the mathematical model. In our measurements we always left the plate floating, which let us use the existing model. The conductor can only maintain a uniform potential, constant in space.

It is important to note that if the load admittance is purely capacitive, as is the case with our interface circuitry for the three-wavelength sensor (see Appendix E), then the voltage ratio response cannot have positive phase for any value of the admittances Y_{11} and Y_{12}, as long as their imaginary part is non-negative, i.e. there is no inductive element. This is important, as it would alert us to a problem if positive phase greater than the noise margin was measured. If positive phase is measured, that will indicate that a leakage path has been established between the sensing electrode and ground. We witnessed this problem when the three-wavelength sensor was placed on a conducting aluminum plate and dust particles penetrated through the Kapton substrate and made contact with the sensing electrode. The problem can be avoided by placing the sensor on an insulating plastic plate, as became our standard practice.

3.4 Testing

3.4.1 Testing in Air

After the coating process, described in Section 3.2, the three-wavelength sensors were tested by performing frequency scans in air. A representative scan is shown in Figure 3-9. The phase angle is near zero for the entire scan and the magnitude of the response remains constant. As is clear from equation 3.28, this is only possible if there is no conduction between the electrodes of the sensor due to contamination.

Another important observation that can be made from the results in Figure 3-9 is that the assumption of infinite input resistance of the interface box, as listed in Table E.1, is indeed valid. More importantly, the test in air is a worst case situation, because if the sensor is used on any kind of material, the input resistance of the interface box would be even more negligible. This results from the following considerations: Since throughout the entire range of frequencies the sensor itself looks purely

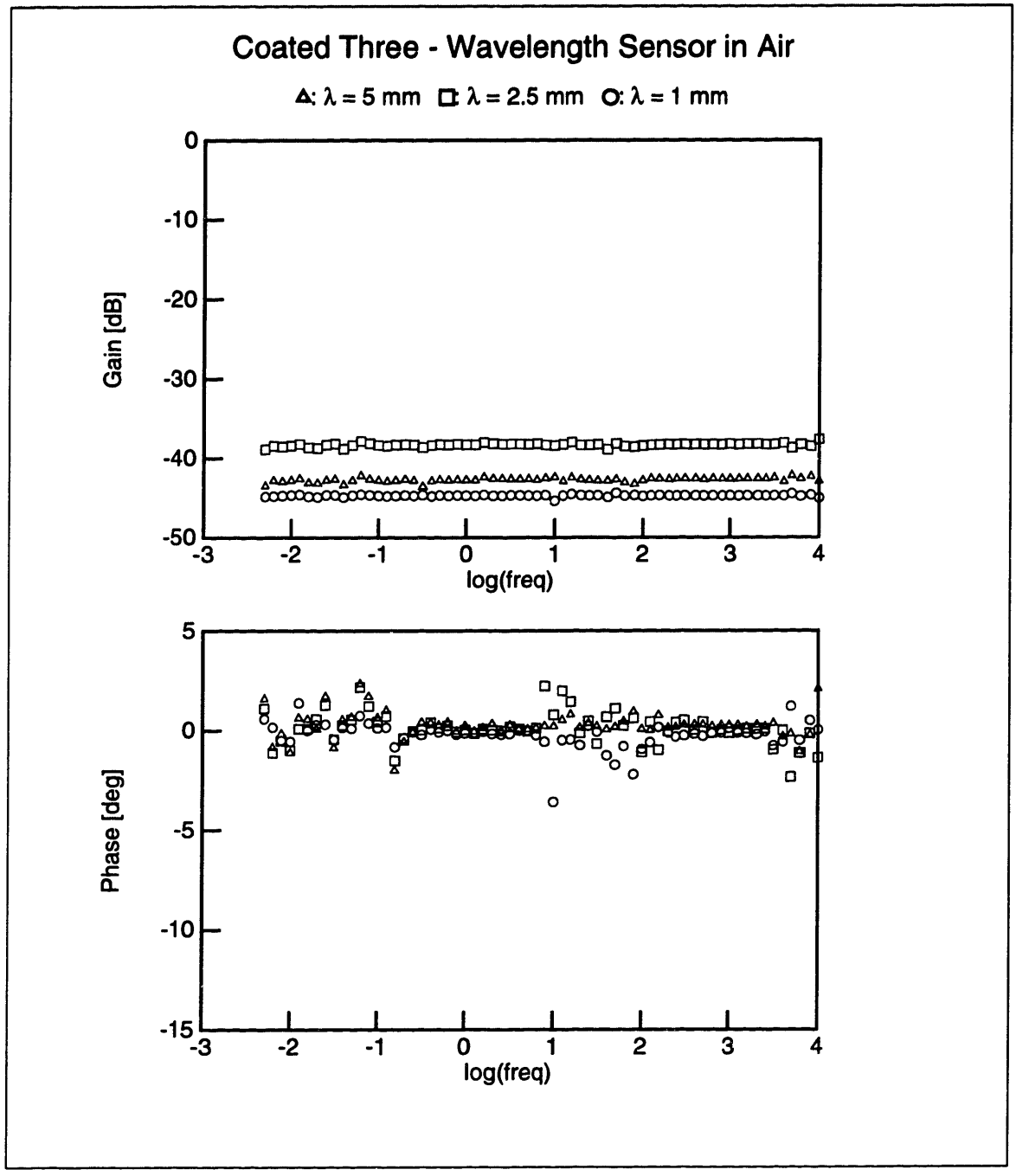

Figure 3-9: A frequency scan of the Parylene coated three-wavelength sensor in air

capacitive, the equation for the voltage ratio becomes

$$\frac{\hat{V}_S}{\hat{V}_D} = \frac{j\omega C_{12}}{j\omega(C_{12} + C_{11} + C_L) + (1/R_L)} \tag{3.29}$$

where we have used equations 3.27 and 3.28 and the following relationships: $Y_{12} = j\omega C_{12}$, $Y_{11} = j\omega C_{11}$, and $Y_L = j\omega C_L + (1/R_L)$. This voltage ratio has a zero at the origin and a pole at $s = -1/[(C_{12} + C_{11} + C_L)R_L]$. Zero phase can result only if the pole occurs at a frequency significantly lower than 0.005 Hz, which is the bottom limit of our frequency range. Since C_{12} is lowest for air, any material with a dielectric constant greater than ϵ_0 would only act to increase this capacitance and push the pole further toward lower frequencies, making its effects more negligible.

In conclusion, the test in air serves to show that the sensor is clean, i.e. there is no parasitic conduction due to contaminants on the electrode surface, and that the input resistance of the interface box can safely be assumed to be infinite for use of the sensor on all materials.

3.4.2 Testing in Transformer Oil

The next step in testing the proper operation of the three-wavelength sensor is to immerse it in a well-known material, such as transformer oil, which is sufficiently conducting to show an appreciable phase angle of the response. The test was performed in Shell Diala A transformer oil. The raw gain-phase data is shown in Figure 3-10.

The appreciable phase angle shows that the conduction in oil is high enough that the parameter estimation would be sensitive to it and could measure it (see Section 2.1.3). An interesting observation can be made in Figure 3-10. It is clear from that plot that all three wavelengths approach the phase peak simultaneously as the excitation reaches lower frequencies. This means that the oil appears to be uniform in the vicinity of the sensor, because the frequency at the phase peak is the same in spite of the different depths of penetration of the three wavelengths.

It is interesting to see the dielectric spectrum that corresponds to the data in Figure 3-10. We discuss the different methods of calculating ϵ^* from M and φ in

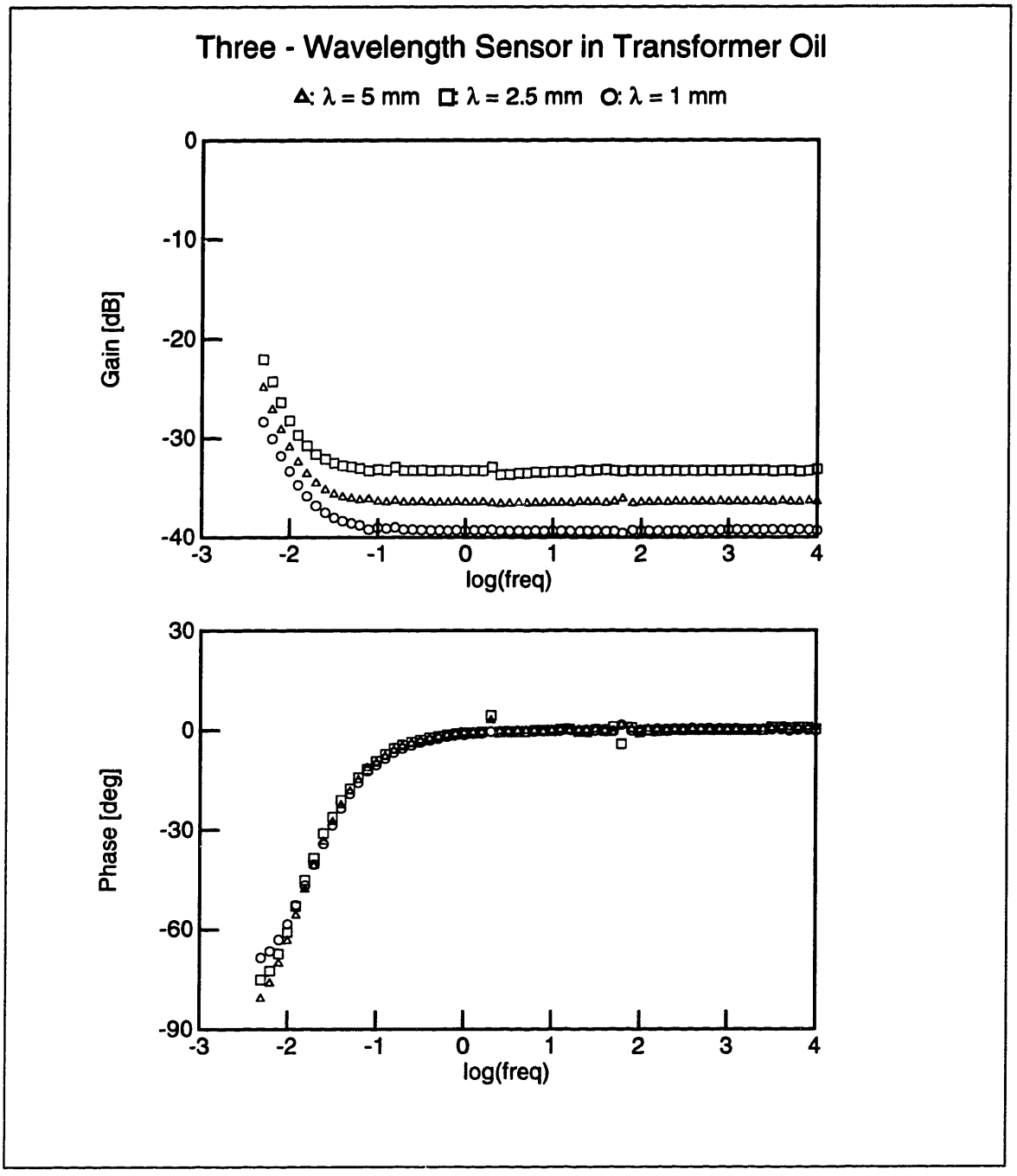

Figure 3-10: Raw gain-phase data of the three-wavelength sensor in Shell Diala A transformer oil

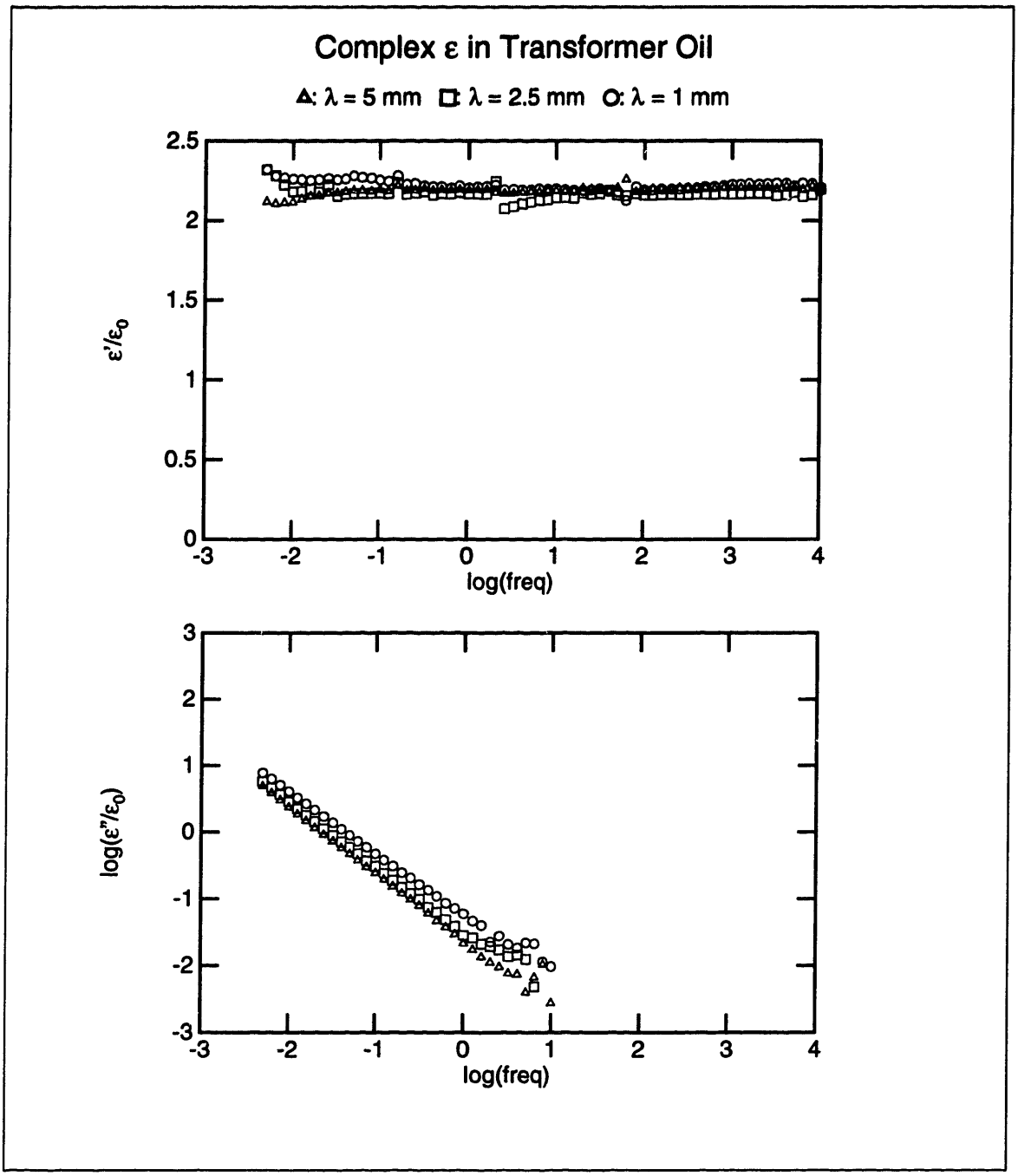

Figure 3-11: Dielectric spectrum of Shell Diala A transformer oil taken with the three-wavelength sensor. The results include all significant spatial Fourier modes.

Chapter 4. If we use the one-dimensional estimation method (Section 4.2), which assumes that the material next to the sensor is uniform to infinity, we obtain the results in Figure 3-11. This figure shows that our conclusion of the uniformity of the oil is not exactly true, because the values of ϵ'' measured by the three wavelengths are a little different. The values of ϵ'' measured by the shortest (1 mm) wavelength are the highest, indicating that the oil was most conducting closer to the sensor. The slope of ϵ'' on the log-log scale of Figure 3-11 is very close to -1, as expected for a material which exhibits *ohmic* behavior, i.e. whose conductivity is independent of the frequency, as already discussed in Section 2.1.2. At low frequencies ϵ' deviates from the constant value due to the presence of a double layer in a way similar to the results in Figure 2-4, which were measured with the parallel-plate sensor.

The average value of the relative dielectric constant in Figure 3-11 is $\epsilon/\epsilon_0 = 2.2$, which is the same value measured with the parallel-plate sensor in Figure 2-4 and the value quoted in the literature. The average conductivity in Figure 3-11 is $\sigma = 1.4 \times 10^{-12}$, which is typical of transformer oil. The self-consistency of these results and their agreement with previously known parameters confirm the proper operation of the three-wavelength sensor.

Chapter 4

Parameter Estimation Algorithms

In Section 3.3 we showed how to solve the *forward* problem of calculating the magnitude and the phase angle of the voltage ratio of an interdigitated sensor from the parameters of the sensor and the properties of the materials above it. What is meant by *parameter estimation* is the reverse problem of finding properties of the materials from measured magnitude and phase data. Numerical techniques must be used for this purpose, since it is impossible to express the solution of the forward problem in closed form. In this chapter we discuss the various techniques for parameter estimation that we have developed and tested.

4.1 Dielectric Profiles and Degrees of Freedom

In Section 4.2 we discussed how some qualitative information about the spatial variations of the dielectric properties of a material can be obtained from the comparison of data taken with interdigitated sensors of several different spatial wavelengths. The next step would be to try to combine the results from all of these wavelengths in a quantitative manner, in order to calculate this spatial variation.

There are many limitations to how much can be learned about the inhomogeneous medium from measurements with the three-wavelength sensor. In this section we investigate the fundamental mathematical limitations, such as the number of degrees of freedom, etc.

4.1.1 Information Contained in Measurements with the Same Wavelength at Different Frequencies

The mechanism by which the frequency of excitation affects the distribution of the electric fields in the material above the electrodes of an interdigitated sensor is that a layer approaches an equipotential surface (i.e. can be approximated as a perfect conductor) for radian frequencies below the relaxation frequency σ/ϵ. The complex admittance of the sensor is a function of the frequency of excitation ω and the complex permittivities ϵ_i^* of the N layers above it, $f(\omega, \epsilon_1^*, \ldots, \epsilon_N^*)$. If we define

$$f_j(\epsilon_1^*, \ldots, \epsilon_N^*) \equiv f(\omega_j, \epsilon_1^*, \ldots, \epsilon_N^*) \qquad \omega = \omega_1, \omega_2, \ldots \tag{4.1}$$

then the functions f_j are mathematically independent of each other, i.e. if set equal to the measured values at these frequencies they would yield a set of equations with a finite number of solutions:

$$f_j(\epsilon_1^*, \ldots, \epsilon_N^*) = Y_j \qquad j = 1, 2, \ldots, N \tag{4.2}$$

where Y_j are the measured values of the sensor's admittance at the frequencies $\omega_1, \omega_2, \ldots, \omega_N$.

This means that, in principle, if the dielectric properties of all materials above the sensor are independent of frequency, then a frequency scan contains all the information about the spatial distribution of the dielectric properties, and the latter may be calculated from the former. Although this statement is true in a strictly mathematical sense, physical considerations impose severe constraints on its validity. As an example, after a layer approaches an equipotential, negligible electric fields will penetrate through it, thereby making the measurement insensitive to the material properties above this layer. The level of detectability of the properties of layers above such a layer quickly decreases with further reduction of the frequency and soon becomes overwhelmed by measurement noise.

Reference [3, pp. 88] summarizes these considerations in Theorem 3.2, which states

that a unique determination of the complex permittivities of all layers is possible only if the relaxation times of the materials are sufficiently distinct and they appear in decreasing order as one moves away from the electrode surface. This constraint makes this approach useful only in specific cases, where the material layers are in the required order.

For the sake of completeness we should mention that the requirement that the dielectric properties be independent of frequency may be relaxed if the dispersive material possesses a universal reference spectrum, from which it deviates in a known manner with changes in some physical parameter. Pressboard, for example, is a dispersive material with such a universal spectrum, as shown earlier in Chapter 2.

4.1.2 Complex Numbers and Degrees of Freedom

For the rest of this section we shall assume that measurements are made at a single frequency with one or more interdigitated sensors of different spatial wavelengths. This means that for every wavelength one value of the complex amplitude of the voltage ratio response $Me^{j\varphi}$ is measured. This complex amplitude includes both gain and phase information. Our goal is to use these different complex amplitudes to calculate some unknown parameters of the medium.

From mathematics we know that in general n independent equations are needed to determine n unknowns "uniquely", i.e. yield a finite number of solutions. This also applies to complex functions of complex variables. We define the number of *degrees of freedom* of a system to be the number of independent equations that relate its unknown parameters.

Before we go on, we must answer the following question: Must we associate one or two degrees of freedom with a complex function? As we shall see in Section 4.5, this question is critical to the parameter estimation method with an assumed profile function.

Any complex function $f(z_1, z_2, \ldots, r_1, r_2, \ldots)$, whose arguments may in general be complex $(z_1, z_2, \ldots)$ or real $(r_1, r_2, \ldots)$, may be represented by two real functions of

real arguments as follows:

$$f(z_1, z_2, \ldots, r_1, r_2, \ldots) =$$

$$= u(x_1, y_1, x_2, y_2, \ldots, r_1, r_2, \ldots) + jv(x_1, y_1, x_2, y_2, \ldots, r_1, r_2, \ldots) \qquad (4.3)$$

where x_i and y_i are the real and imaginary components of the complex numbers z_i:

$$z_i = x_i + jy_i \qquad (4.4)$$

The equation

$$f(z_1, z_2, \ldots, r_1, r_2, \ldots) = 0 \qquad (4.5)$$

is therefore equivalent to the two equations

$$u(x_1, y_1, x_2, y_2, \ldots, r_1, r_2, \ldots) = 0 \qquad (4.6)$$

$$v(x_1, y_1, x_2, y_2, \ldots, r_1, r_2, \ldots) = 0 \qquad (4.7)$$

Two degrees of freedom should be assigned to a complex equation, *two* degrees of freedom are necessary for the determination of every complex unknown z_i, and *one* degree of freedom would be necessary for every real unknown r_i.

There are, however, some special cases when these rules for determining the necessary number of equations do not apply and a greater number of equations is required. Let us suppose that two or more real unknowns can be "lumped" together in every complex equation into "clusters", which are real functions of these unknowns. These clusters need not be the same in every equation, but they should involve the same set of unknowns. All clusters in any one given equation must be the same and none of the cluster variables may appear outside of a cluster. Symbolically, this may be expressed in the following way: The system of equations

$$f_i(z_1, z_2, \ldots, r_1, r_2, \ldots) = 0 \qquad i = 1, 2, \ldots \qquad (4.8)$$

could in this case be written as

$$f_i'[z_1, z_2, \ldots, g_i(r_1, \ldots, r_m), r_{m+1}, \ldots] = 0 \qquad i = 1, 2, \ldots \tag{4.9}$$

where the functions g_i of the real variables $r_1, \ldots, r_m$ are real. In the solution of this system of equations, after all of the other variables have been eliminated, we are left with a set of equations of the following form

$$g_i(r_1, \ldots, r_m) = c_i \qquad i = 1, 2, \ldots \tag{4.10}$$

where each of these equations is the result of *one* equation from the previous set (4.8). This means that for the unique determination of the unknowns $r_1, \ldots, r_m$ each of them needs to be assigned *two* degrees of freedom, since one complex equation is needed per variable. This need for extra information comes about because some information is lost in the requirement that the constants c_i, which are in general complex numbers, must be real in order to match g_i. In other words, some redundancy is present in the original set of equations 4.8, which need to be such that the coefficients c_i are real. If they do not yield real c_i, then the system of equations will have no solution. A simple example that illustrates this principle is presented in Appendix F.

This special case, where more degrees of freedom than expected are necessary, occurs in the use of an assumed profile function, discussed in Section 4.5.

In the documentation of the parameter estimation programs, listed in Appendix H, a different nomenclature is used, which assigns only one degree of freedom to every complex equation and variable. The mathematics implemented by this software uses the correct number of degrees of freedom, as defined in this section.

4.1.3 Analytic Functions of Complex Variables

A complex function $f(z) = u(x,y) + jv(x,y)$, where $z = x + jy$, is *analytic* if its derivative can be uniquely defined [8, sec. 10.4]. In other words, the expression

$$\frac{d}{dz}f(z) = \lim_{\Delta z \to 0} \frac{f(z+\Delta z) - f(z)}{\Delta z} \tag{4.11}$$

should be independent of the direction in the complex plane from which Δz approaches zero. Formally, we need:

$$\begin{aligned} \frac{d}{dz}f(z) &= \lim_{|\Delta z| \to 0} \frac{f(z+\Delta z) - f(z)}{\Delta z} \\ &= \lim_{\Delta x \to 0} \frac{f(z+\Delta x) - f(z)}{\Delta x} = \lim_{\Delta y \to 0} \frac{f(z+j\Delta y) - f(z)}{j\Delta y} \end{aligned} \tag{4.12}$$

If we then write

$$\lim_{\Delta x \to 0} \frac{f(z+\Delta x) - f(z)}{\Delta x} = \frac{\partial u}{\partial x} + j\frac{\partial v}{\partial x} \tag{4.13}$$

$$\frac{1}{j} \cdot \lim_{\Delta y \to 0} \frac{f(z+j\Delta y) - f(z)}{\Delta y} = \frac{1}{j}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial y} \tag{4.14}$$

and equate the real and imaginary parts of equation 4.12 after substituting equations 4.13 and 4.14 into it, we obtain the *Cauchy-Riemann Equations* [8]:

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{4.15}$$

$$\frac{\partial v}{\partial x} = -\frac{\partial u}{\partial y} \tag{4.16}$$

We can conclude that an analytic function cannot have a purely real argument r, because equations 4.15 and 4.16 would yield $\partial f/\partial r = 0$. Of course we are free to assign only real values to a complex argument of an analytic function. The real and imaginary parts u and v of an analytic function are related to each other via equations 4.15 and 4.16.

Most functions that can be expressed in closed form, such as polynomials, rational functions, exponentials, trigonometric functions, etc., are analytic. Sums, products, ratios, compound functions, etc. of analytic functions yield analytic functions, etc. Examples of functions that are not analytic include $\Re\{z\}$, $\Im\{z\}$, and complex conjugation. The complex gain expressed as a function of the ϵ^* of the materials above the sensor, derived in Section 3.3, is an analytic function, because the operations in every step of the process of finding the admittance as a function of the complex permittivities are such that the analytic character of the function is preserved. This is an important result for the multidimensional parameter estimation routine, presented in Section 4.4.

4.2 One-Dimensional Parameter Estimation

This is the simplest kind of parameter estimation in which only one parameter is unknown, namely the complex permittivity ϵ^* of one of the layers above the interdigitated sensor. The routine takes data from *one* wavelength and uses a root-finding algorithm to find the unknown parameter. The gain and phase of the response are a function of the unknown ϵ^*, implemented by the routine **gp()**, which is discussed in detail in Section 3.3 and listed in Section G.4. The problem is that of finding a root (zero) of the difference of this function and the measured results.

The Secant method [9] is used for this root-finding method. We do not include a listing of the code that implements this function, because it has not been translated into C. Instead, we have been using the already compiled version of this program, written in FORTRAN by Dr. M. Zaretsky and listed in [3]. Its name is **parestso** and it can be found on the computer LEES-OMEGA-K.

An arbitrary number of known layers may be included along with the unknown layer, i.e. no unnecessary assumptions need to be made about the structure of layers, other than the assumption that the unknown layer's dielectric properties do not change with variations in x (see Figure 3-5).

Since the interdigitated sensors have a limited depth of penetration, if the unknown

layer extends to infinity (i.e. beyond the reach of the longest wavelength), the sensors would be sensitive only to the properties of the material adjacent to the sensor. If the material is indeed homogeneous, then sensors of any spatial wavelength would measure the same value of ϵ^*. If, however, ϵ^* depends on x, then the value of ϵ^* measured by an interdigitated sensor would be some sort of a weighted average, with the depth of sampling proportional to the spatial wavelength λ of the sensor. Consequently if a material is not homogeneous, the three parts of the three-wavelength sensor would measure different values of ϵ^*.

Although this method can show homogeneity, if the material is inhomogeneous it can only provide a qualitative picture of how ϵ^* varies with x. It is nevertheless a very useful tool and is probably the first step to take when interpreting dielectric profile data. Chapter 5 shows many instances of the application of this method to data from measurements.

4.3 Marching Approach

This is the first method which attempts to combine the results from more than one wavelength into a quantitative description of the spatial dielectric profile of an inhomogeneous medium. Its iterative algorithm is based on a series of one-dimensional estimations of the kind described in Section 4.2 and thus avoids the complications associated with multidimensional searches [3].

This method approximates the dielectric profile of the structure above it by a stair-step function, with the intervals of this function being determined by the program itself. It is therefore only applicable to the problem of finding an approximation to the dielectric profile of one single unknown layer extending to infinity, as no *a priori* information may be specified about the widths of the different regions.

An assumption is made that every sensor of spatial wavelength λ has a depth of penetration into the material equal to $\alpha\lambda$, where α is a parameter which reflects the assumed discreteness of the regions.

Let us suppose we have N sensors of distinct spatial wavelengths λ_i, $\lambda_1 < \lambda_2 <$

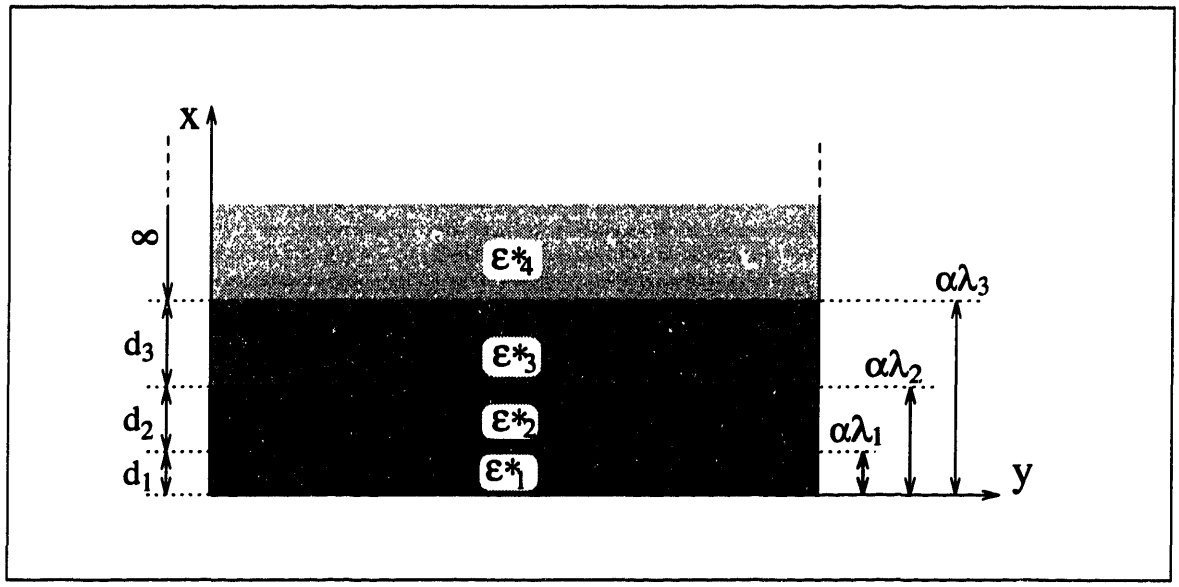

Figure 4-1: Stair-step approximation of a dielectric profile with the marching approach

$\ldots < \lambda_N$. We discretize the medium of continuous dielectric properties above the sensors with n homogeneous layers of thickness $d_1 = \alpha\lambda_1$, $d_2 = \alpha\lambda_2 - d_1$, $\ldots$, $d_N = \alpha\lambda_N - \sum_{i=1}^{N-1} d_i$. This division is shown in Figure 4-1. Each of the N layers is characterized by a complex permittivity ϵ_i^*. The method finds the values of ϵ_i^* that would yield the measured gain and phase response.

We are assuming that we can use the algorithm of Section 4.2 to calculate the complex permittivity of a single unknown layer from the measurement with a single wavelength. At first we assume that the bottommost layer extends to infinity, i.e. only one layer with a complex permittivity of ϵ_1^* exists. We then apply the one-dimensional search to the data taken with the shortest spatial wavelength λ_1, since according to our initial assumption it is sensitive only to the first region. In the next step two regions are assumed; the first is the bottommost layer of thickness d_1, whose complex permittivity is assumed to be known and equal to the previously calculated value of ϵ_1^*; and the second layer is assumed to extend to infinity and be uniform with a permittivity of ϵ_2^*. Then we apply the one-dimensional parameter estimation method to the measurement with the sensor of spatial wavelength λ_2, which results in a value for ϵ_2^*. After that we use the measurement with the sensor of wavelength λ_3 and

apply it to an assumed structure with three layers, the top unknown layer extending to infinity and characterized by a complex permittivity of ϵ_3^*, and so on until the data from all wavelengths has been used. This way we end up with the first approximation to the dielectric profile which concludes the first iteration step.

For the second iteration step we follow a similar procedure to the one outlined above, but instead of assuming that the unknown layers extend to infinity we give them the assumed thickness as specified in Figure 4-1 and apply the one-dimensional search to the structure which includes all layers. If ϵ_i^* of layer number i is being estimated from the data taken with the sensor of spatial wavelength λ_i, then all layers below the current one, i.e. with index numbers less than i, use the values for ϵ^* from the current iteration, while all layers above i use the values of ϵ^* calculated in the previous iteration. The second iteration also begins with estimating ϵ_1^* and ends with the topmost layer.

The third iteration is identical to the second one, etc. The iterations continue until subsequent iterations stop changing the calculated values of ϵ^* for all layers.

The parameter α represents the assumed reach of a given wavelength. Its value is chosen according to two criteria. In the first place, it should be influenced by the actual dielectric profile so as to provide the best stair-step fit to it. For example, if the complex permittivity of the inhomogeneous medium changes very quickly close to the surface of the electrodes, a smaller value for α would result in a better fit, since all of the assumed layer thicknesses would be smaller. Conversely, if the material is roughly uniform, larger values of α would result in a better fit. In the second place, α needs to be such that the method will converge: too large values of α may fail to provide a close enough approximation to the profile function and lead to no convergence.

The parameter α usually takes up values between 0.1 and 0.5. Applying this method to an exponential profile and plotting the least squares error of the resulting stair-step fit versus α showed that best results are obtained for $\alpha = 0.25$[1].

The greatest advantage of this method lies in its relative simplicity in that it avoids

[1]Personal correspondence with Dr. P. A. von Guggenberg, Doble Engineering, Watertown, Massachusetts, USA.

multidimensional searches by only solving for one unknown at a time. This also makes it somewhat more robust in terms of convergence than the multidimensional searches. However, this method does not allow for the specification of structures with layers of arbitrary width or the inclusion of known layers. The marching approach is perfectly fitted to situations where a single inhomogeneous material is in intimate contact with the interdigitated electrodes and its thickness is greater than the reach of the longest wavelength.

4.4 Multi-Dimensional Parameter Estimation

This is a process which searches for more than one unknown variable simultaneously. It is useful in cases when there are more than one unknown layer in a material structure and data from more than one spatial wavelength is available. There are no limitations to the thicknesses or the position and order of the unknown layers.

An inhomogeneous layer may be approximated by a number of unknown homogeneous layers forming a stair-step function in a way similar to the marching approach of Section 4.3. However, in this case these sublayers may be assigned arbitrary thicknesses in a way that would approximate the profile function more closely.

4.4.1 A Root-Finding Algorithm

In this method we are looking for exact solutions for the complex permittivity of the unknown layers. As discussed in Section 4.1, two degrees of freedom are necessary for every unknown layer and two degrees of freedom are assigned to every spatial wavelength, which means that the number of unknown layers must be equal to the number of spatial wavelengths. The three-wavelength sensor can therefore be used to measure the complex permittivities of three unknown homogeneous layers above it.

Every measurement with a specific spatial wavelength λ_i creates one complex equation of the form

$$f_i(\epsilon_1^*, \epsilon_2^*, \ldots) = gp_i(\epsilon_1^*, \epsilon_2^*, \ldots) - M_{mi}^* = 0 \tag{4.17}$$

where the function *gp* represents the forward process of evaluating the complex magnitude of the voltage ratio described in Section 3.3 and M_m^* is the measured value of the complex gain. This is how one equation in the set of equations results from every spatial wavelength.

The root-finding algorithm is based on a hybrid between the Newton-Raphson and the Secant methods [9] as follows: After an initial guess is made, the new guesses are calculated on the basis of the old ones via the following recursive formula:

$$X_{n+1} = X_n + \Delta X_n = X_n - J^{-1} Y_n \tag{4.18}$$

where $X = [\epsilon_1^*, \epsilon_2^*, \ldots, \epsilon_N^*]$, is the vector of unknown variables, and $Y = [f_1, f_2, \ldots, f_N]$ is the result vector of applying the functions f_i from equation 4.17 to the current values in X for every wavelength. The index number n refers to the number of the iteration. In equation 4.18 J^{-1} is the inverse of the *Jacobian* matrix J, which is defined as

$$J = \begin{bmatrix} \partial f_1/\partial \epsilon_1^* & \partial f_1/\partial \epsilon_2^* & \cdots & \partial f_1/\partial \epsilon_n^* \\ \partial f_2/\partial \epsilon_1^* & \partial f_2/\partial \epsilon_2^* & \cdots & \partial f_2/\partial \epsilon_n^* \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial \epsilon_1^* & \partial f_n/\partial \epsilon_2^* & \cdots & \partial f_n/\partial \epsilon_n^* \end{bmatrix} \tag{4.19}$$

The Jacobian matrix is defined because the complex functions f_i of complex variables ϵ_i^* are analytic (see Section 4.1). In order to calculate any one complex partial derivative in equation 4.19, only two real partial derivatives are necessary, since the other two are given by equations 4.15 and 4.16. If the functions had not been analytic, it would have been necessary to split every unknown complex variable into two real variables and every complex equation into two equations. In principle, this can be done even if the functions are analytic, but this would double the amount of work in calculating the Jacobian, since we would not be taking advantage of equations 4.15 and 4.16. Since all derivatives are calculated numerically, this would significantly increase the amount of computation. In order to confirm that the functions f_i are indeed analytic, we performed numerical differentiation of all four partial derivatives

that comprise a complex derivative and the results did indeed satisfy the Cauchy-Riemann equations 4.15 and 4.16.

In the hybrid method the Jacobian matrix and its inverse are not calculated for every iteration, but the old matrix is used for several iterations before a new one is computed. A new Jacobian is calculated also if more than five damping steps (described below) are taken in one iteration, since that would indicate that J is out of date.

Since the Newton-Raphson and the Secant methods may become unstable and severely deviate from a root, it is necessary to introduce *damping* to the algorithm. After a new guess is computed, a test is performed to determine whether the new guess is closer to the root than the old one. A vector X is considered closer to the root of the system of equations if the absolute value of its corresponding result vector Y is smaller. If the new guess is not closer, then a half step backward is taken, i.e. instead of letting $X_{n+1} = X_n + \Delta X_n$, we let $X_{n+1} = X_n + \frac{1}{2}\Delta X_n$. If this guess is still worse than the old one, a quarter step back is taken, etc. If more than five such steps are needed, we conclude that the correction vector ΔX_n needs to be updated. Also, if this vector is already up to date, but still damping does not get a better guess, the method fails and no solution is found.

Another kind of damping may be needed to avoid converging to non-physical roots, e.g. complex permittivities that would correspond to negative conductivities or to dielectric constants less than ϵ_0. If such a condition is detected, a similar strategy as above is applied in which the algorithm goes back and halves the correction vector recursively until the new set of values is valid.

Since in the process of finding new values for the unknown parameters the inverse of the Jacobian matrix is calculated, this matrix must be non-singular, which is similar to the requirement that the slope be non-zero in the one-dimensional Newton-Raphson method. This means that at least one unknown layer must fall within the reach of every sensor and every unknown layer must fall within the reach of at least one sensor. Although this is a necessary requirement to find a solution, it might not be a sufficient condition for a non-singular matrix. We have not, however, been able

Initial guess		True ϵ_1^*		Final result		#	Time
ϵ_1	σ_1	ϵ_1	σ_1	ϵ_1	σ_1		[m:s]
8.85×10^{-12}	0	8.85×10^{-12}	10^{-12}	8.81×10^{-12}	9.99×10^{-13}	5	1:48
8.85×10^{-12}	10^{-11}	8.85×10^{-12}	10^{-12}	8.85×10^{-12}	1.00×10^{-12}	8	4:37
5×10^{-11}	0	8.85×10^{-12}	10^{-12}	8.85×10^{-12}	1.00×10^{-12}	7	3:08
8.85×10^{-12}	0	3×10^{-11}	0	3.00×10^{-11}	0.0	5	1:57
8.85×10^{-12}	10^{-11}	3×10^{-11}	0	3.00×10^{-11}	-2.35×10^{-17}	9	4:13
5×10^{-11}	0	3×10^{-11}	0	3.00×10^{-11}	0.0	7	3:28

Table 4.1: Computation time of program **est.c** as a function of initial guess and solution using multidimensional estimation. The computation was performed on a XENIX 80486-based 33 MHz machine.

to find a situation when the method would fail due to such a problem.

The iterations stop either when the absolute value of the result vector is less than a prespecified tolerance, or when the new iteration yields a new guess which is very close to the old one.

Convergence case studies of this method indicated that convergence is always reached if every unknown layer is well within the scope of at least one sensor. We applied this method to computer-generated data, which simulated two layers with various values of the conductivity and the permittivity and with various initial guesses. Convergence was reached in all cases. Table 4.1 lists the parameters of these tests, as well as the total number of iterations # that were necessary to find the root. We also tested the behavior of this estimation process if noise was added to the computer-generated data. The results of these tests are shown in Table 4.2. Noise at the input naturally resulted in noise in the output, but did not seem to affect the ability of the algorithm to find solutions. It is conceivable that instrumentation errors inherent in every measurement might cause the problem to have no solution. This obstacle could then be overcome by allowing for a larger tolerance in the convergence test. The negative values for the conductivity in Table 4.2 are small variations about zero due to noise.

As mentioned earlier, a new Jacobian matrix is calculated every several iterations. The root-finding method is closer to the Newton-Raphson method if a new Jacobian is computed more often, while if J is rarely updated the method is closer to the Secant

Index	↓	Input		Output	
		Gain [dB]	Phase [deg]	ϵ	σ
No noise	1	-38.30	-43.68	8.85×10^{-12}	0.0
	2	-41.40	-57.98	8.85×10^{-12}	1.0×10^{-12}
	1	**-36**	-43.68	1.36×10^{-11}	6.35×10^{-11}
	2	-41.40	-57.98	8.54×10^{-12}	9.76×10^{-13}
	1	-38.30	**-45**	8.72×10^{-12}	1.03×10^{-14}
	2	-41.40	-57.98	8.88×10^{-12}	9.98×10^{-13}
	1	-38.30	-43.68	7.63×10^{-12}	-9.64×10^{-14}
	2	**-40**	-57.98	1.09×10^{-11}	1.17×10^{-12}
	1	-38.30	-43.68	9.10×10^{-12}	-2.66×10^{-14}
	2	-41.40	**-60**	8.31×10^{-12}	1.02×10^{-12}

Table 4.2: Effect of noise on the results from the multidimensional parameter estimation. The input values in bold are the ones that have been altered.

Frequency of updates	1	2	3	5
Computation time [min]	4	9	11	16

Table 4.3: Computation time versus frequency of Jacobian updates for **est.c**. A frequency of N means that a new Jacobian is calculated every N iterations. The computation was performed on a XENIX 80486-based 33 MHz machine.

method. The former converges much faster, but the calculation of the Jacobian costs a lot of extra computation time. These are two competing factors in terms of computation time cost. In order to test for the optimal frequency of Jacobian updates, we applied this root-finding algorithm with the same input but with different values for this frequency. The results, shown in Table 4.3, indicate that for this set of data, convergence was reached fastest if a new Jacobian was calculated at every step, in spite of all the extra computation associated with this. However, if there is a large number of unknowns this may no longer be true, since the number of computations associated with finding the Jacobian increases with the square of the number of unknowns.

The code for this algorithm is listed in Appendix H. The main program is called **est.c** and it is listed in Section H.4. All of the subsidiary routines listed in Section H.5 are part of the program. It also uses all of the auxiliary routines, such as those listed in Sections H.7 and H.6. A sample input file is listed in Section H.8.1.

4.4.2 An Optimization Algorithm

If we have more sensors than unknown layers the problem becomes overspecified, i.e. the set of equations will in general have no solution. However, the extra information should in principle contribute to finding an even closer approximation to the profile function. Therefore the problem is one of *optimization*, i.e. finding the set of values for all of the unknown complex permittivities that would minimize the set of functions f_i defined in equation 4.17. The function that we are trying to minimize is given as the sum of the squares of all of the individual functions:

$$F = \sum_{i=1}^{M} |f_i(\epsilon_1^*, \epsilon_2^*, \ldots)|^2 = \sum_{i=1}^{M} |gp_i(\epsilon_1^*, \epsilon_2^*, \ldots) - M_{mi}^*|^2 \tag{4.20}$$

In theory, if the data is error-free, such as computer-generated data, then the problem would have a solution even if it is overspecified, because the extra measurements would be redundant. However, unlike the case of equal number of degrees of freedom, adding experimental noise to an overspecified problem will lead to no solution. This is when the optimization method becomes extremely useful. Adding more wavelengths would act to reduce the effects of measurement noise.

Optimization techniques have the additional advantage that they tend to be much more stable than root-finding methods. They do have one major drawback, though: there may be many local minima, and so one can never be sure that the global minimum has been found. It is therefore of a crucial importance to begin with a guess very close to the true solution, as illustrated in Chapter 5.

Powell's Method

This numerical technique of multidimensional optimization is based on a series of one-dimensional optimizations. It is described in great detail in [9]. An original set of N directions in the N-dimensional variable space are chosen and one-dimensional minimizations along these directions are performed sequentially. During this process the set of directions are updated in a way that brings them closer to pointing toward the minimum. A theorem [9] states that after at most N^2 one-dimensional

minimizations, the minimum is reached.

The one-dimensional minimization used by Powell's method is Brent's method [9], which uses several different algorithms depending on which of them is more appropriate, to yield a highly efficient minimization algorithm.

The code for our implementation of this minimization technique is listed in Section H.4. Its name is **estm.c** and it is based on the same set of subsidiary and auxiliary routines as the program **est.c**, described in the previous subsection. A sample input file to this program is listed in Section H.8.1.

Simplex Method

A *simplex* is a body of $n + 1$ vertices in n-dimensional space, whose n-dimensional volume is not zero. In one-dimensional space it is a line segment of non-zero length; in two-dimensional space it is a triangle of non-zero area; in three-dimensional space it is a tetrahedron of non-zero volume.

In this method an initial simplex undergoes a series of transformations, such as reflection, shrinking, and expansion, based on the values of the function F at its vertices. The algorithm is such that the simplex moves toward a minimum in an attempt to surround it and then shrink around it until its volume falls below a prespecified tolerance.

The code for this routine may be found in Section H.4, together with all of the other estimation routines, under the name of **ests.c**. A sample input file is included in Section H.8.1.

Case studies showed that this program takes longer to arrive at the solution than the root-finding program **est.c**, as shown in Table 4.4. The input to the optimization routine for the cases shown in Table 4.4 is the same as in Table 4.1. The number of iterations in Table 4.4 does not have to correspond to the number of iterations in Table 4.1, because these are very different algorithms. The limitations of the optimization algorithm are clearly visible in Table 4.4, where the third and fourth instance give the wrong answer, because they arrive at a different local minimum than the one corresponding to the true solution, and the fifth instance terminated

Initial guess		True ϵ_1^*		Final result		#	Time
ϵ_1	σ_1	ϵ_1	σ_1	ϵ_1	σ_1		[m:s]
8.85×10^{-12}	0	8.85×10^{-12}	10^{-12}	8.43×10^{-12}	1.01×10^{-12}	72	10:31
8.85×10^{-12}	10^{-11}	8.85×10^{-12}	10^{-12}	8.67×10^{-12}	1.01×10^{-12}	166	23:27
5×10^{-11}	0	8.85×10^{-12}	10^{-12}	4.89×10^{-11}	2.01×10^{-12}	55	8:10
8.85×10^{-12}	0	3×10^{-11}	0	1.10×10^{-11}	-1.05×10^{-13}	117	16:42
8.85×10^{-12}	10^{-11}	3×10^{-11}	0	8.85×10^{-12}	1.0×10^{-11}	0	0:40
5×10^{-11}	0	3×10^{-11}	0	3.00×10^{-11}	-7.18×10^{-14}	74	10:51

Table 4.4: Computation time of program **ests.c** as a function of initial guess and solution using the simplex method. The computation was performed on a XENIX 80486-based 33 MHz machine.

prematurely even before the first iteration, because the values of the optimization function at the vertices of the simplex were too close together. Of course, there are cases when the root-finding technique is not applicable, as discussed earlier for the case of too many wavelengths.

Either optimization routine can be faster than the other under the right set of circumstances. In general, which of the routines in this section should be used for a particular problem depends on what is known about the problem. In some cases both the root-finding technique and an optimization method could be used to gain confidence in a result.

4.5 Assumed Profile Function Estimation

So far we have approximated a dielectric profile by stair-step functions. This is the simplest and a very general kind of approximation. It makes it possible to use the existing models, which assume that the medium above the interdigitated electrodes consists of a set of homogeneous layers to approximate an inhomogeneous layer.

These methods provide no means of using *a priori* knowledge about the functional form of the dielectric profile in the parameter estimation. If we know that a function has a specific form, then we might be able to find this function exactly with only a limited number of degrees of freedom. If we know nothing about a function, then we would need an infinite number of degrees of freedom to determine it exactly, because

only an infinite set of equations would let us decrease the width of the intervals in the stair-step function to zero. A simple mathematical example illustrating this principle is shown in Appendix F.

In this section we investigate the functional form of dielectric profiles in pressboard, which result from moisture mass transfer processes. We already know from Chapter 2 how the complex permittivity of pressboard depends on its moisture content. What remains to be seen is what profiles the moisture content in pressboard may assume.

4.5.1 Diffusion Equation

We are only considering diffusion in one dimension. This is justified by the fact that we are interested in modeling processes that occur in high-power transformers, where the pressboard appears in thin sheets, with one surface in contact with the oil. The thickness of these sheets is very small compared to their other dimensions and therefore we may assume that the moisture content shows no variations with y or z, but depends only on x (see Figure 4-1 for axis definition). In this case the diffusion of water in pressboard is governed by the following equation:

$$\frac{\partial m}{\partial t} = \frac{\partial}{\partial \xi}\left[D(m)\frac{\partial m}{\partial \xi}\right] \tag{4.21}$$

where m is the moisture concentration and $D(m)$ is the diffusion coefficient, which is, in general, a function of the moisture. This fact introduces a non-linearity in equation 4.21 which would make it very difficult to solve. Since our goal is to obtain the general form of the functional dependence, we shall assume from now on that $D(m)$ is just a constant. Since there may be other layers, such as the Parylene coating, between the pressboard sheet and the electrodes, it is convenient to define a spatially shifted variable ξ such that $\xi = 0$ at the exposed outer surface of the pressboard, and $\xi = d$ at the sealed surface, where d is the thickness of the pressboard sample. If the distance between the exposed surface and the sensor is x_1, then $\xi = x_1 - x$ (see Figure 5-1).

The closed-form solutions to equation 4.21 are

$$m(\xi,t) = e^{-Dk^2t}\sin k\xi \tag{4.22}$$

$$m(\xi,t) = e^{-Dk^2t}\cos k\xi \tag{4.23}$$

$$m(\xi,t) = \text{erf}\left(\frac{L-\xi}{2\sqrt{Dt}}\right) \tag{4.24}$$

$$m(\xi,t) = \text{erf}\left(\frac{L+\xi}{2\sqrt{Dt}}\right) \tag{4.25}$$

where k is the diffusion equation separation constant and L is an arbitrary parameter. These solutions are pairwise independent, i.e. either pair of equations may be used to find the total solution of a diffusion problem. The *error function* $\text{erf}(x)$ is defined as follows [8]:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-\eta^2}d\eta \tag{4.26}$$

and it has the bell curve as its derivative. In most cases infinite sums of these functions are needed to match boundary conditions and particular solutions. The first two forms are more convenient at times longer than the characteristic diffusion time

$$\tau = \frac{d^2}{D} \tag{4.27}$$

while the last two forms are more convenient at times shorter than τ, when a change at one surface has not had the time to propagate to the other.

Let us assume that the exposed surface, i.e. the surface of the pressboard in contact with the oil, is constrained by the equilibrium with oil (see Section 1.3) to be at a certain constant concentration, and that this concentration experiences a step change at time $t = 0$ such that

$$m(\xi = 0, t < 0) = m_0 \tag{4.28}$$

$$m(\xi = 0, t > 0) = m_1 \tag{4.29}$$

We are also assuming that all previous transients in the moisture distribution have had time to die away such that at $t = 0^-$ the moisture is uniformly distributed, i.e.

$$m(\xi, t = 0^-) = m_0 \tag{4.30}$$

At the other surface there is no moisture flux, since it is sealed by the sensor. The boundary condition that corresponds to this situation is

$$\frac{\partial m}{\partial \xi}(\xi = d, t) = 0 \tag{4.31}$$

We recognize that out of the solutions to the diffusion equation, only equation 4.22 satisfies these boundary conditions, for values of k given as:

$$k_n = \frac{n\pi}{2d} \qquad\qquad n = 1, 3, 5, \ldots \tag{4.32}$$

A sum of terms in the form of equation 4.22 and with the above values for k constitute the homogeneous part of the solution to this differential equation. The particular solution is simply

$$m_p(\xi, t) = m_1 \tag{4.33}$$

Since at time $t = 0$ the moisture distribution is

$$m(\xi, t = 0) = \begin{cases} m_1 & \xi = 0 \\ m_0 & \xi > 0 \end{cases} \tag{4.34}$$

we must find an infinite series with terms in the form of equation 4.22, which would converge to this function. At $t = 0$ the exponential part of equation 4.22 is unity. We may therefore use a Fourier sine series expansion to write

$$u(\xi) = \sum_{n=1,\text{odd}}^{\infty} \frac{4}{\pi n} \sin k_n \xi \tag{4.35}$$

with k_n defined as

$$k_n = \frac{\pi n}{2d} \tag{4.36}$$

Finally we may write the total solution to the differential equation:

$$m(\xi, t) = m_1 + (m_0 - m_1) \sum_{n=1,\text{odd}}^{\infty} \frac{4}{\pi n} e^{-Dk_n^2 t} \sin k_n \xi \tag{4.37}$$

Figure 4-2 shows a family of functions in the form of equation 4.37, every curve corresponding to a specific value of normalized time t', defined as:

$$t' = \frac{\pi^2 Dt}{4d^2} \tag{4.38}$$

As is clear from equation 4.37, higher modes die out with time at a considerably higher rate as compared to lower modes due to the factor k_n^2 in the exponential. At large values of the normalized time one can see that only the fundamental spatial mode of the transient is present, which is simply a one-quarter period of a sine in space.

Based on equation 4.37 we may assume that the moisture concentration in the pressboard sample at some instant of time is of the functional form

$$m(\xi) = A + B \sum_{n=1,\text{odd}}^{\infty} \frac{4}{\pi n} e^{-Dtk_n^2} \sin k_n \xi \tag{4.39}$$

4.5.2 Profile Functions

Equations 2.36 and 2.37 show how the dielectric spectrum of pressboard varies with changes in its moisture content. We repeat these equations here for convenience:

$$\log\left(\frac{\epsilon' - \epsilon_\infty}{\epsilon_0}\right) = \mathcal{F}'\{\log\omega - [f_T(T) + f_M(m)]\} \tag{4.40}$$

$$\log\left(\frac{\epsilon''}{\epsilon_0}\right) = \mathcal{F}''\{\log\omega - [f_T(T) + f_M(m)]\} \tag{4.41}$$

For the moisture contents of interest, i.e. for contents less than about 3%, the

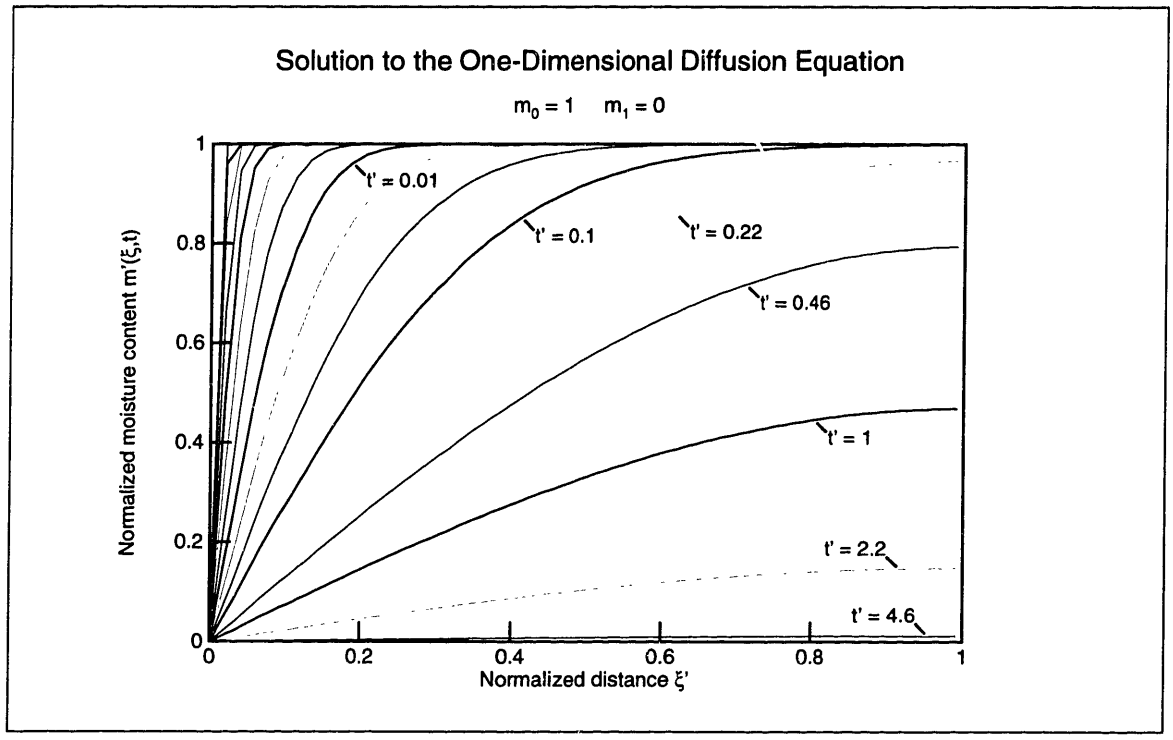

Figure 4-2: Solutions to the diffusion equation at different values of normalized time $t' = (\pi^2 Dt)/(4d^2)$

loss peak in pressboard occurs at frequencies below 0.005 Hz (see Section 2.3.1), and the only part of the dielectric spectrum visible is the decrease to the right of the loss peak, which on a log-log plot is a straight line of slope $\gamma \approx -0.7$ (see Figure 2-14). We may therefore write

$$\mathcal{F}'(x) = c_1 + \gamma x \tag{4.42}$$

$$\mathcal{F}''(x) = c_2 + \gamma x \tag{4.43}$$

where c_1 and c_2 are constants. Before we are ready to write the general equation relating ϵ^* to the moisture content m, we need to know the function $f_M(m)$, which represents the logarithmic frequency shift as a function of the moisture content. As can be seen in Figure 2-17, little can be said about this function. Therefore we choose

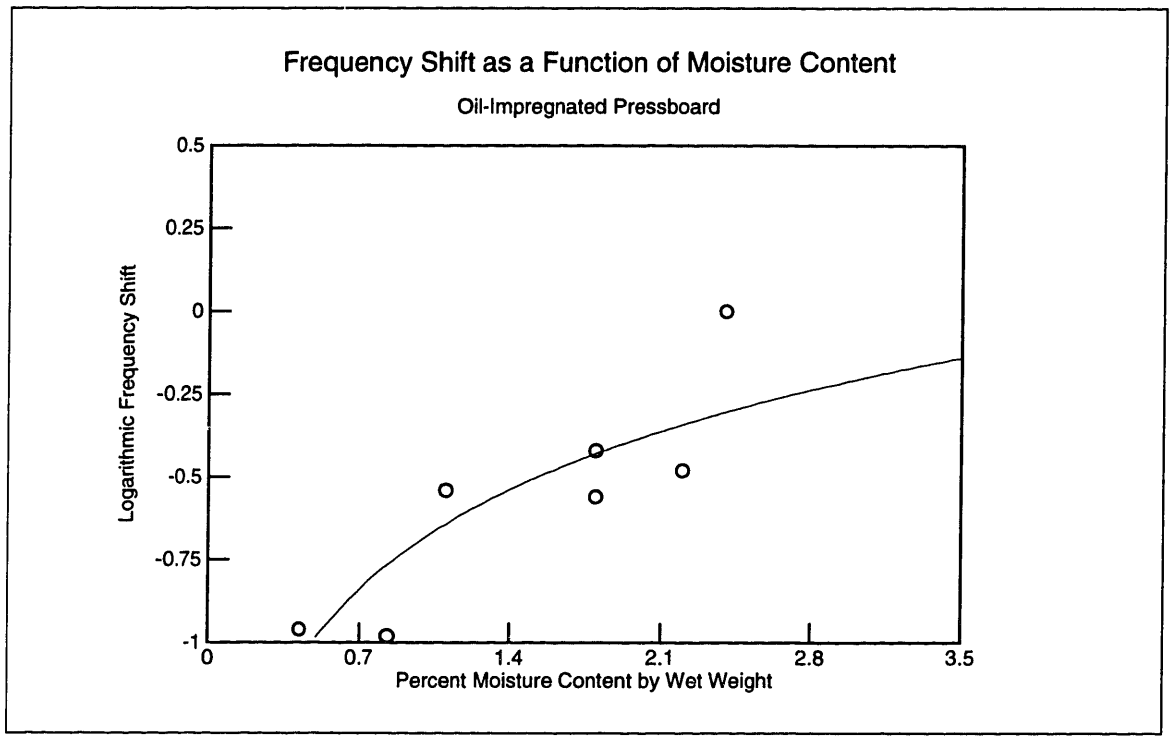

Figure 4-3: Curve fitting of equation 4.44 to the data representing the frequency shift as a function of moisture

a form that would yield the simplest algebra, namely

$$f_M(m) = c_m + \log m \tag{4.44}$$

whose best fitted curve to the data in Figure 2-17 is shown in Figure 4-3. The curve in this figure corresponds to $c_m = -0.684$, if m is in percent and the frequency is in Hertz.

Combining equations 4.40, 4.41, 4.42, 4.43, and 4.44 yields the final functional form:

$$\epsilon^* = \epsilon_\infty + \epsilon_0 c \left(\frac{m}{\omega}\right)^{-\gamma} \tag{4.45}$$

in which c is a complex coefficient and the factor ϵ_0 is added to normalize this complex coefficient. Now we are ready to substitute equation 4.39 into equation 4.45 to obtain

the general form of the dielectric profile of pressboard:

$$\epsilon^*(\xi) = \epsilon_\infty + \epsilon_0 c \left[\frac{1}{\omega}\left(A + B\sum_{n=1,\text{odd}}^{\infty}\frac{4}{\pi n}e^{-Dtk_n^2}\sin k_n\xi\right)\right]^{-\gamma} \tag{4.46}$$

There is a problem with this formulation, because there are too many coefficients: c, A, and B. We could factor out the coefficient A and introduce the two new coefficients $c^* = cA$ and $C = B/A$, but this would cause a problem: It is conceivable that the moisture distribution has no constant term, i.e. $A = 0$, which can result from an initial condition of $m_0 = 0$. This condition would yield $C = \pm\infty$ and $c^* = 0$, a very problematic situation for numerical computations. For this reason we use the following relationship:

$$A + BX = \left(A + \frac{B}{2}\right)\left[1 + \frac{2B}{2A+B}\left(X - \frac{1}{2}\right)\right] \tag{4.47}$$

to define the following new set of parameters:

$$c^* = c\left(A + \frac{B}{2}\right)^{-\gamma} \tag{4.48}$$

$$C = \frac{2B}{2A+B} \tag{4.49}$$

The physical requirement of non-negative moisture content imposes the constraints $A \geq 0$ and $A - B \geq 0$. Therefore $(A + B/2)$ is non-negative and raising it to the $-\gamma$ power is possible. In addition to that the problem of infinite coefficients has been solved. This revision results in the final form of the profile function:

$$\epsilon^*(\xi) = \epsilon_\infty + \epsilon_0 c^* \left[\frac{1}{\omega}\left(1 - \frac{C}{2} + C\sum_{n=1,\text{odd}}^{\infty}\frac{4}{\pi n}e^{-Dtk_n^2}\sin k_n\xi\right)\right]^{-\gamma} \tag{4.50}$$

The range of physical values for C, i.e. values for which the moisture content is non-negative, is $C \in [-2, 2]$, which covers all allowed ratios of the coefficients A and B in 4.46. For example, $A = 0$ corresponds to $C = 2$; $B = 0$ corresponds to $C = 0$; and $B = -A$, which is the lowest bound for B, corresponds to $C = -2$.

The dielectric profile is fully specified by the three coefficients: c^*, C, and Dt. If these coefficients are found by a parameter estimation method, then the dielectric profile of the pressboard, and therefore its moisture profile, are fully specified. The first of these coefficients is complex (c^*), but the other two (C and Dt) are real, for a total of four degrees of freedom. Data from two spatial wavelengths would not be enough to determine these coefficients, though, because C and Dt form a real "cluster", as defined in Section 4.1. Therefore two degrees of freedom are required for each of them, for a total of six. Data from three spatial wavelengths is needed to solve for the parameters of this profile function. The three-wavelength sensor is therefore perfectly suited for this task.

4.5.3 Parameter Estimation

The model developed in Section 3.3 assumed several discrete layers of material with sudden changes in ϵ^* at the interfaces. Laplace's equation (3.11) was satisfied everywhere except at these interfaces, because ϵ^* was constant. Now that we have a case where ϵ^* is a function of space, we will have to develop a new model. We begin with a direct consequence of conservation of charge:

$$\vec{\nabla} \cdot \epsilon^* \vec{\tilde{E}} = 0 \tag{4.51}$$

$$\vec{\nabla}\epsilon^* \cdot \vec{\tilde{E}} + \epsilon^* \vec{\nabla} \cdot \vec{\tilde{E}} = 0 \tag{4.52}$$

$$\vec{\nabla}\epsilon^* \cdot \vec{\nabla}\tilde{\Phi} + \epsilon^* \nabla^2 \tilde{\Phi} = 0 \tag{4.53}$$

where the tildes ($\tilde{}$) represent complex amplitudes as defined in equation 3.1.

From Section 3.3, equation 3.7, we know that

$$\tilde{\Phi}(x,y) = \sum_{n=-\infty}^{\infty} \hat{\Phi}_n(x) e^{-jk_n y} \tag{4.54}$$

which lets us convert equation 4.53 to a full differential equation for every Fourier

mode n:

$$\frac{d\epsilon^*}{dx} \cdot \frac{d\hat{\Phi}_n}{dx} + \epsilon^* \left(\frac{d^2\hat{\Phi}_n}{dx^2} - k_n^2 \hat{\Phi}_n \right) = 0 \tag{4.55}$$

Equation 4.55 has a closed-form solution if ϵ^* is an exponential function of x, but for simplicity we will take ϵ^* to be piece-wise constant so that we can use the old standard model, where $d\epsilon^*/dx = 0$ and 4.55 reduces to Laplace's equation, but with a much greater number of homogeneous layers, so that the profile function 4.50 can be approximated as closely as we want. Unlike the root-finding technique of Section 4.4.1, where the number of layers in the stair-step approximation was limited by the number of degrees of freedom, in this case there is no limit (other than computation time) to the number of layers in the stair-step approximation, because the number of unknowns remains three: c^*, C, and Dt. If we then define the unknown vector X to be $X = [c^*, C, Dt]$, with Y still defined as $Y = [f_1, f_2, f_3]$, with f_i defined in equation 4.17, we may use the root-finding technique developed in Section 4.4.1 to find the three unknown parameters using data taken with the three-wavelength sensor. It could also be possible to develop the optimization counterpart of this method, in analogy to the techniques presented in Section 4.4.2.

The code for this algorithm is listed in Appendix H. The main program is called **estp.c** and it is listed in Section H.4. All of the subsidiary routines listed in Section H.5 are part of the program. It also uses all of the auxiliary routines, such as those listed in Sections H.7 and H.6. A sample input file is listed in Section H.8.1.

Chapter 5

Profile Measurements

5.1 Experimental Setup

In order to measure moisture profiles in pressboard, the sample had to be placed in a controlled environment, which allowed moisture to diffuse in and out of the pressboard from one surface. The other surface of the sample was sealed by the sensor itself. For this purpose the experimental setup shown in Figure 5-1 was created. The stainless steel chamber can be filled with transformer oil, whose moisture content can be varied by bubbling wet or dry nitrogen through it, or the chamber can remain full of air with controlled pressure and humidity, as was done in the experiments presented in this chapter. Since intimate contact between the pressboard and the sensor needs to be maintained, the sample has to be tightly squeezed from both sides. The teflon and aluminum layers serve this purpose, while at the same time allowing mass-transfer processes to occur at that surface through a multitude of holes. They are attached to the aluminum base with insulating nylon bolts.

5.2 Oil-Free Pressboard under Vacuum

In order to test the experimental setup shown in Figure 5-1, we placed a sample of oil-free pressboard in it and took continuous full-range frequency scans while the air in the chamber was being evacuated. The data was taken only with one spatial wavelength,

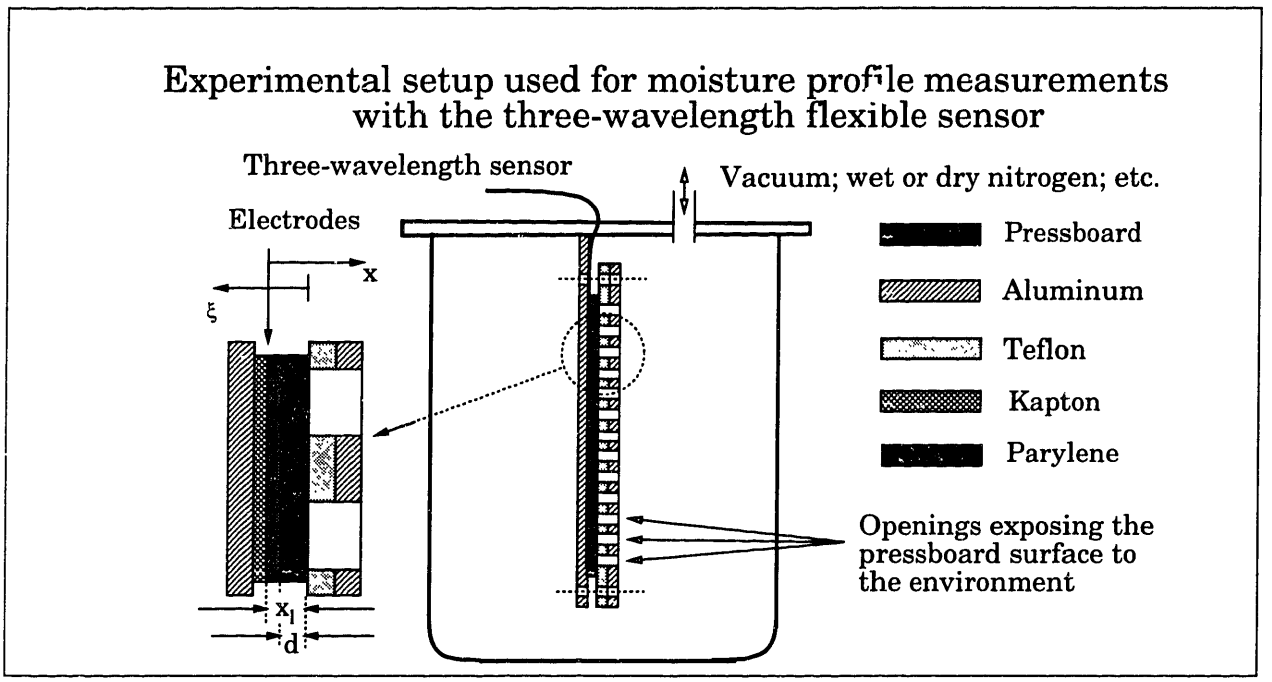

Figure 5-1: Experimental setup for profile measurements taken with the 3-λ sensor

namely 1 mm. The one-dimensional parameter estimation algorithm, described in Section 4.2, was used to calculate the complex permittivity of the pressboard.

The results are shown Figure 5-2. The bold lines in this figure correspond to a frequency scan taken before the vacuum was applied. When exposed to the ambient air, oil-free pressboard acquires an equilibrium value of the moisture content of about 5%. At such water concentrations pressboard is relatively conducting, as can be seen in the figure. Two loss peaks are clearly visible in Figure 5-2, the first one at about 50 Hz, and the second one below 0.005 Hz. The dielectric spectra exhibit the behavior predicted by the Kramers-Krönig relations (see Figure 1-2 in Section 1.2.2), with an elevation of ϵ' corresponding to the loss peak in ϵ'', and a relatively flat region of ϵ' between about 0.05 Hz and 1 Hz, corresponding to the interval between loss peaks.

In order to understand the first spectrum taken after vacuum was applied, we must be aware of the fact that the measurements at frequencies above 0.1 Hz happen very quickly, in about five minutes, while the measurements below that frequency take up the rest of the time of the one-hour-long scan. For example, twenty minutes are needed to get the last three points of the sixty-four-point curve. Most of the data

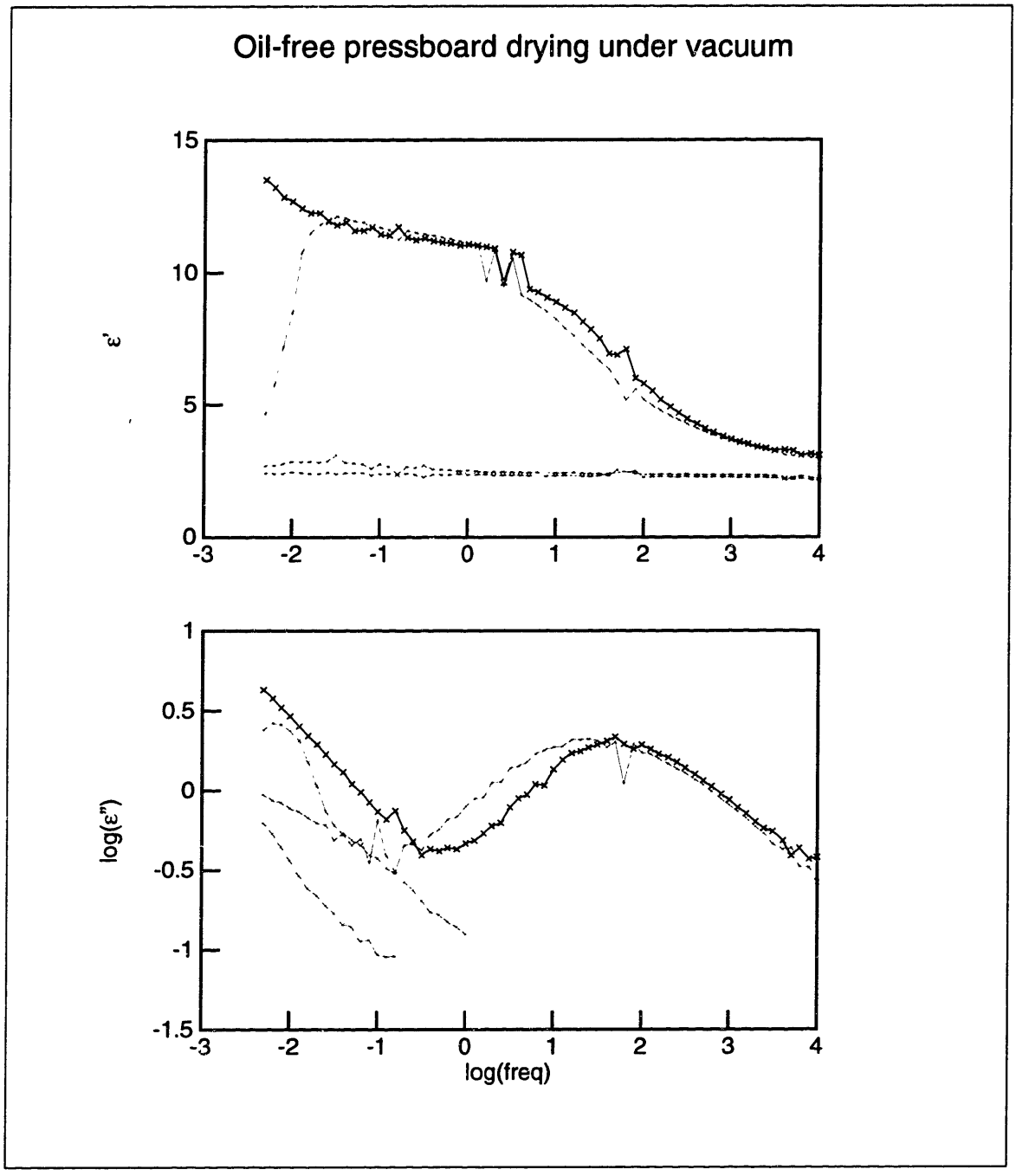

Figure 5-2: Dielectric spectra of oil-free pressboard under vacuum. The bold line is the spectrum measured before vacuum was applied. The other three spectra shown were taken at one-hour intervals after that.

Layer	Material	Thickness [mm]	Permittivity
0	Polymethyl Methacrylate	12.3	3.12 ϵ_0
1	Low Density Polyethylene	0.066	2.26 ϵ_0
2	High Density Polyethylene	0.028	2.26 ϵ_0
3	Parylene	0.005	2.70 ϵ_0

Table 5.1: Layer structure for polymer experiment. Permittivity data is taken from [4] at 1 kHz and room temperature.

points of the first frequency scan under vacuum were taken before water had had the chance to leave the pressboard. However, the curve takes a plunge for frequencies below 0.03 Hz, because at that time the moisture content and the conductivity of the pressboard were already considerably lower.

The second and third spectra after vacuum was applied show the spectrum had shifted to the left as moisture left the pressboard. Looking at the ϵ'' curve, we can only see the right-side leg of the loss peak, which used to be at 50 Hz, but shifted left by more than five units of logarithmic frequency for the second spectrum, and six units for the third spectrum. The spectra for scans taken after the third hour of vacuum are not shown, because the conductivity was so low that it was obscured by noise.

The conductivity of dry oil-free pressboard is too small to measure (under about 5×10^{-14} U/m), while the conductivity of dry oil-impregnated pressboard is still measurable, as shown in Chapter 2. This indicates that the oil makes a major contribution to the conduction of oil-impregnated pressboard.

5.3 Polymers

Before using the three-wavelength sensor to measure profiles in pressboard, we decided to test it on polymers. We substituted the pressboard layer in Figure 5-1 with three layers of polymers as listed in Table 5.1.

Figure 5-3 shows the gain and phase of the response of the three-wavelength sensor. The phase angle remains close to zero for the entire frequency range, indicating

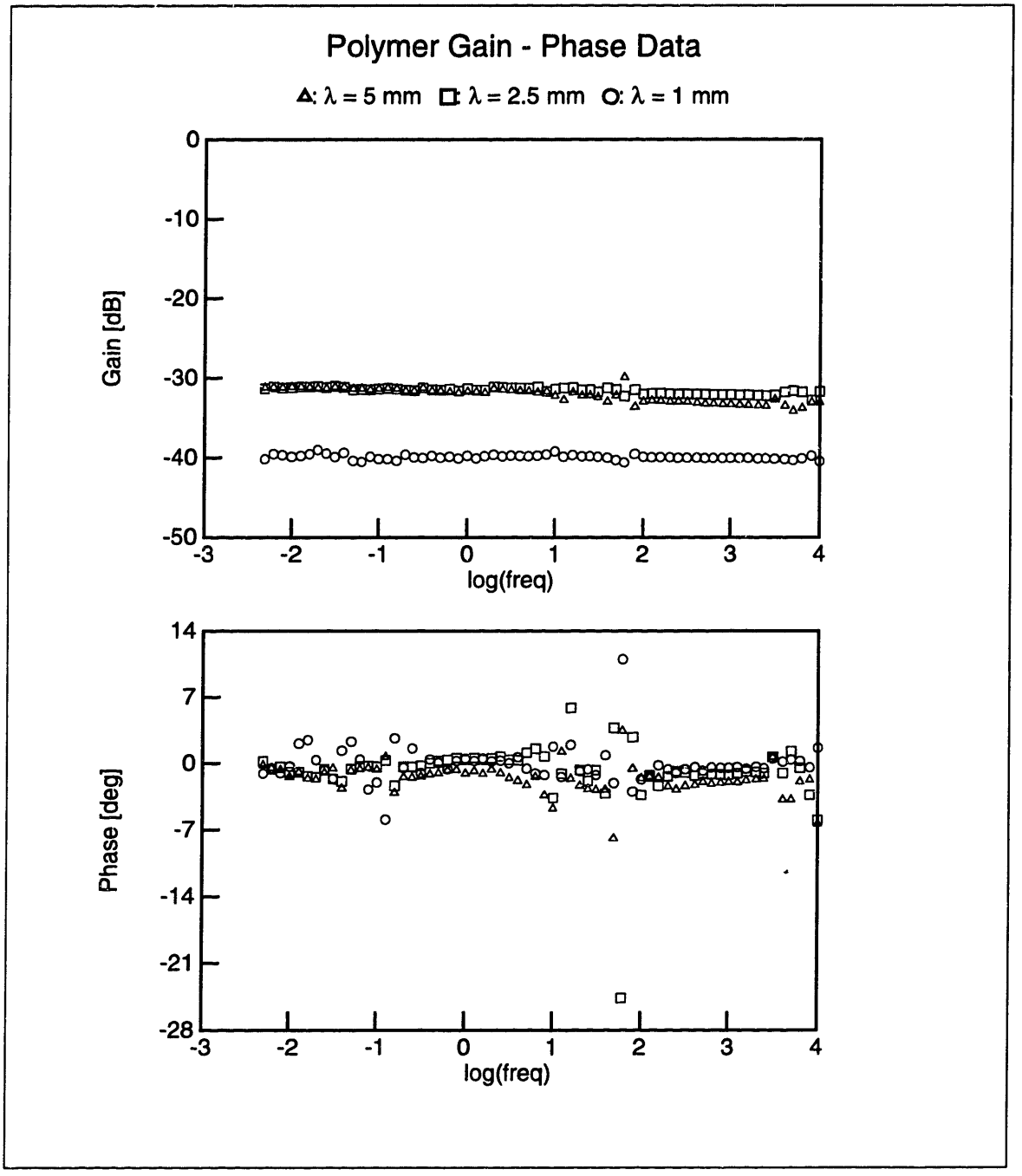

Figure 5-3: Gain-phase data taken with the three-wavelength sensor on polymers

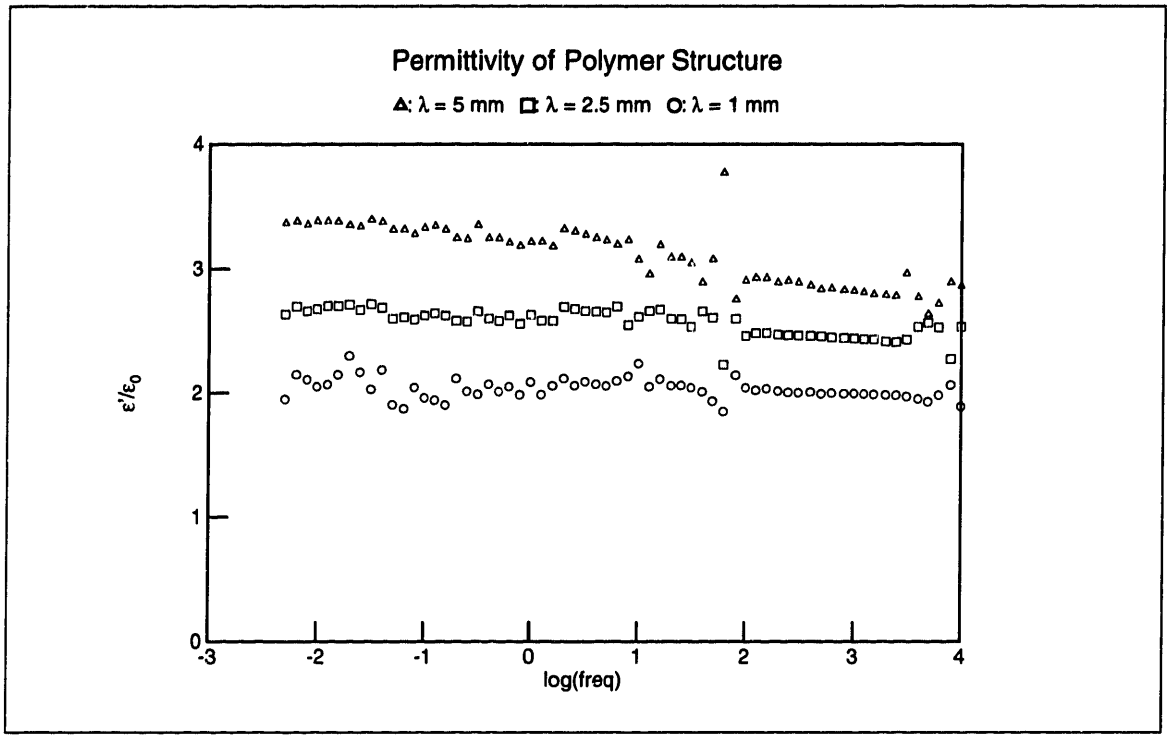

Figure 5-4: Permittivity of polymer structure as calculated from every wavelength of the three-wavelength sensor

that the conductivities of these materials are too low to make a contribution. Using the one-dimensional parameter estimation routine (Section 4.2) we estimated the properties of the polymer layer for every wavelength independently, in order to obtain a qualitative picture of the permittivity distribution. The results are shown in Figure 5-4. Only ϵ' data is shown because the conductivity was too low to measure.

The longest wavelength in Figure 5-4 measured the highest value of the permittivity, indicating that the topmost Plexiglas layer had a higher permittivity than the other two materials, which is consistent with the permittivities of these plastics, shown in Table 5.1 [4]. The effective depth of penetration of an interdigitated sensor is typically one quarter of the spatial wavelength [3] [2]. The reach of the shortest wavelength, $\lambda_3 = 1$ mm (Table 3.1), is therefore about 0.25 mm. The combined width of layers 1 and 2 is less than 0.1 mm, which means that all three wavelengths reach the Plexiglas layer. The thickness of the Plexiglas layer is about ten times larger than the depth of penetration of the longest wavelength, and may be assumed to be

Layer	Assumed thickness	Permittivity
0	∞	1.73 ϵ_0
1	0.066 mm	14.3 ϵ_0
2	0.028 mm	0.48 ϵ_0

Table 5.2: Poor results of applying the root-finding multidimensional parameter estimation algorithm to polymer data at 1 kHz

infinite.

The measurement in Figure 5-4 is consistent with the values of the permittivity of the materials listed in Table 5.1. The Plexiglas layer shows some dispersion, since the permittivities increase with lower frequencies. The shortest wavelength displays the least pronounced dependence on ω, because it is least sensitive to the properties of the Plexiglas.

The data in Figure 5-4 seems to be more noisy than other dielectric spectra presented so far. This is due to the fact that the phase angle of the response is close to zero and the noise compensating influence of having two pieces of data per measurement is absent. The inversion algorithms can only rely on magnitude data, which has a typical measurement tolerance of 0.5 **dB**, corresponding to 6% noise.

When we applied the root-finding multidimensional parameter estimation algorithm (Section 4.4.1) to the data at 1 kHz, the results listed in Table 5.2 were obtained. This set of values for the permittivities resulted in a better than 1% fit of the calculated magnitudes to the measured magnitudes. Nevertheless the values yielded by this method are not very realistic. Although the method worked correctly, the results are poor, because the spatial dielectric profile seen by every wavelength is not sufficiently distinct. All three wavelengths were primarily influenced by the Plexiglas layer and as a result the signal-to-noise ratio of the method with respect to the other two layers is quite low. For good results the unknown layers' thicknesses should be of the same order of magnitude as the depths of penetration of the different wavelengths, so that the shortest wavelength is most sensitive to the closest layer, etc. Some of these ideas were discussed in Section 4.3. Therefore the three-wavelength sensor should work best on 0.5–2 mm thick layers.

For the polymer experiment described in this section, the inversion algorithm that yielded the most information was the one-dimensional parameter estimation.

5.4 Oil-Impregnated Paper

Preliminary measurements with the three-wavelength sensor on EHV-Weidmann HIVAL pressboard exhibited inconsistency in the values of the permittivity between the three wavelengths and the known properties of pressboard (from Chapter 2). We realized that because of the textured surface of that kind of pressboard, an effective oil layer was formed between the pressboard surface and the Parylene coating. Although such a layer can easily be included in the model, it would introduce another unknown and make the measurements less sensitive to the properties of the pressboard instead.

This is why we chose to conduct our profile measurements with Crocker paper, which is a very similar cellulose insulating material with much the same applications as pressboard. The Crocker paper samples had a very smooth surface, which eliminated the problem of the extra oil layer. In addition to that, the smooth surface of the paper makes it possible to stack many layers without worrying about empty space left between the plies. The Crocker paper sample, with which we conducted the experiments discussed in this section, was 0.25 mm thick. Sixteen plies of paper added up to a total thickness of 4 mm, which is enough to warrant the approximation of infinite thickness. The paper was impregnated with Shell Diala A transformer oil by the regular oil-impregnation procedure described in Appendix C.

First we examined the oil-impregnated Crocker paper sample with the parallel-plate sensor, described in detail in Section 2.1. The dielectric spectrum of the sample was taken at room temperature. It is shown in Figure 5-5. This figure is included mainly as a sanity check reference for all subsequent measurements with the three-wavelength sensor on Crocker paper.

The dielectric spectrum of the Crocker paper sample in Figure 5-5 differs from the spectrum of HIVAL pressboard (Figure 2-14) in that in addition to the dominant loss peak, which occurs at frequencies below 0.005 Hz and is present in both materials,

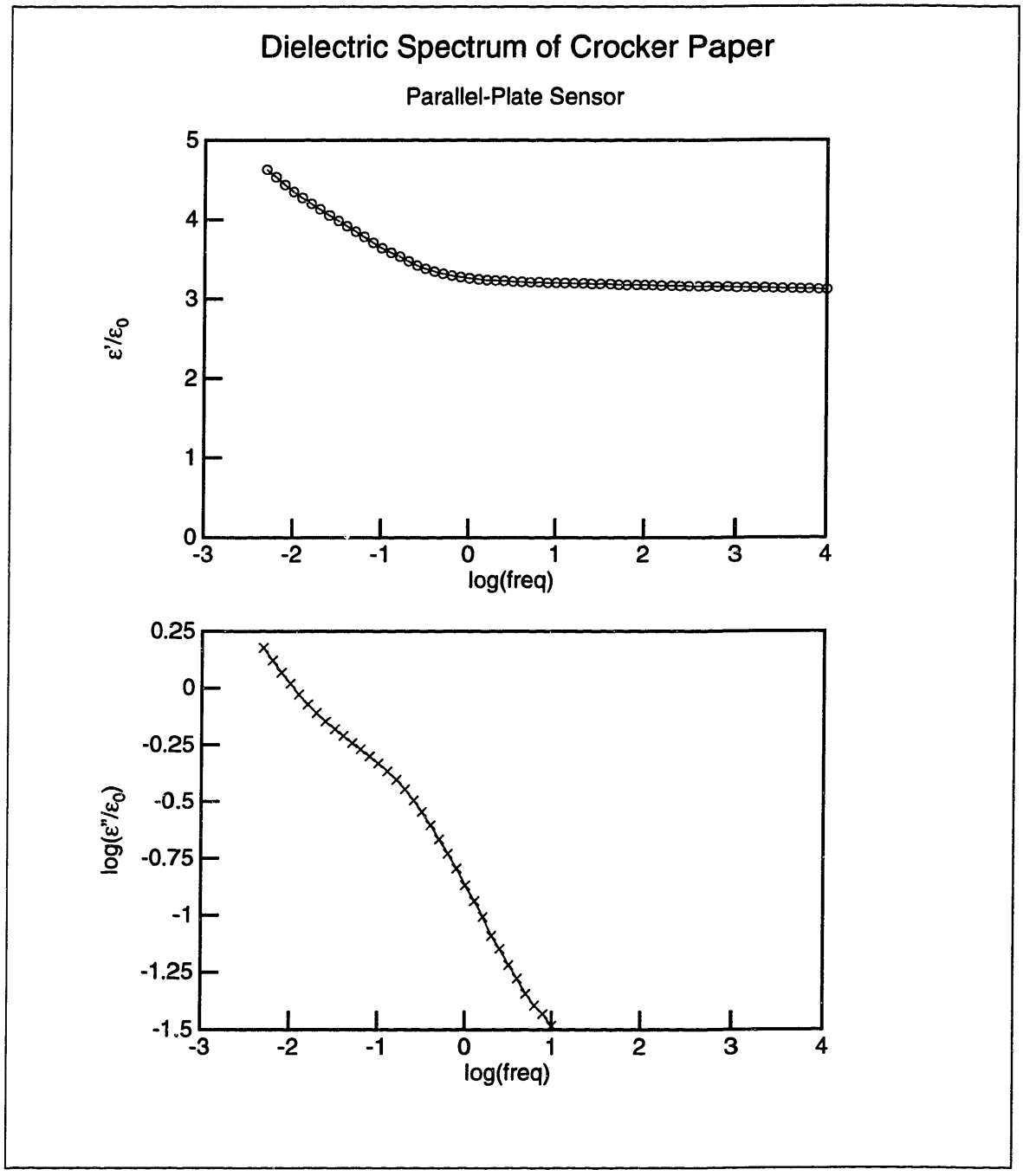

Figure 5-5: Dielectric spectrum of oil-impregnated 0.25 mm Crocker paper at room temperature

Layer	Thickness	Permittivity	Conductivity
0	∞	1.55 ϵ_0	7.49×10^{-13}
1	0.25 mm	3.29 ϵ_0	-1.50×10^{-12}
2	0.25 mm	5.41 ϵ_0	2.32×10^{-12}

Table 5.3: Results from applying the root-finding multidimensional algorithm to Crocker paper data at 0.01 Hz

another loss peak is visible at about 0.2 Hz. This minor peak is responsible for the curved shape of the ϵ'' plot and the point of inflection present in the ϵ' curve between 0.01 and 0.1 Hz.

The first set of measurements with the three-wavelength sensor on the sixteen-ply Crocker paper structure was conducted in air immediately after the oil-impregnated samples had been dried under vacuum. The gain-phase data from this measurement is shown in Figure 5-6. The results from the one-dimensional parameter estimation algorithm, applied to the data from all three wavelengths with the assumption of a single unknown homogeneous layer, are shown in Figure 5-7. The shortest wavelength measured the highest value of ϵ'', suggesting that the layer closest to the sensor is very highly conducting as compared to the bulk of the paper.

This high conductivity near the surface of the sensor may be attributed to the absorption in the paper of moisture which had been adsorbed on the sensor surface.

We also applied the root-finding multidimensional search to the data at 0.01 Hz. The results are shown in Table 5.3. The values of the permittivities are in the order expected from looking at Figure 5-7, i.e. layer 2 had the highest value. It is disturbing to see a negative value of the conductivity of layer 1. This phenomenon has a simple explanation. The dielectric relaxation time of layer 2, which is closest to the sensor, is

$$\tau_e = \frac{\epsilon}{\sigma} = \frac{4.79 \times 10^{-11}}{2.32 \times 10^{-12}} = 20.6 \text{ sec} \tag{5.1}$$

corresponding to a relaxation frequency of

$$f_e = \frac{1}{2\pi\tau_e} = 0.0077 \text{ Hz} \tag{5.2}$$

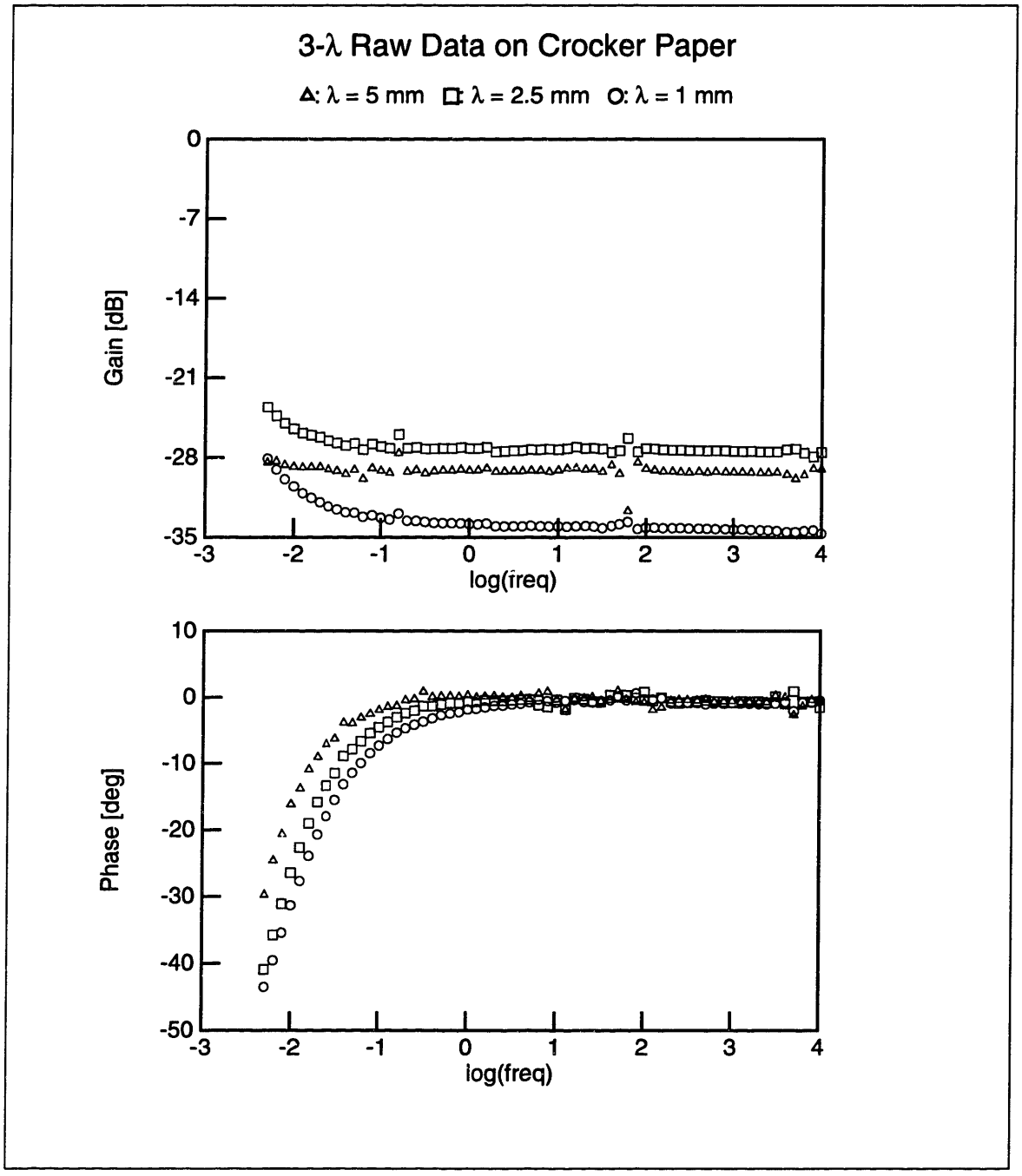

Figure 5-6: Raw gain-phase data taken with the three-wavelength sensor on sixteen-ply Crocker paper

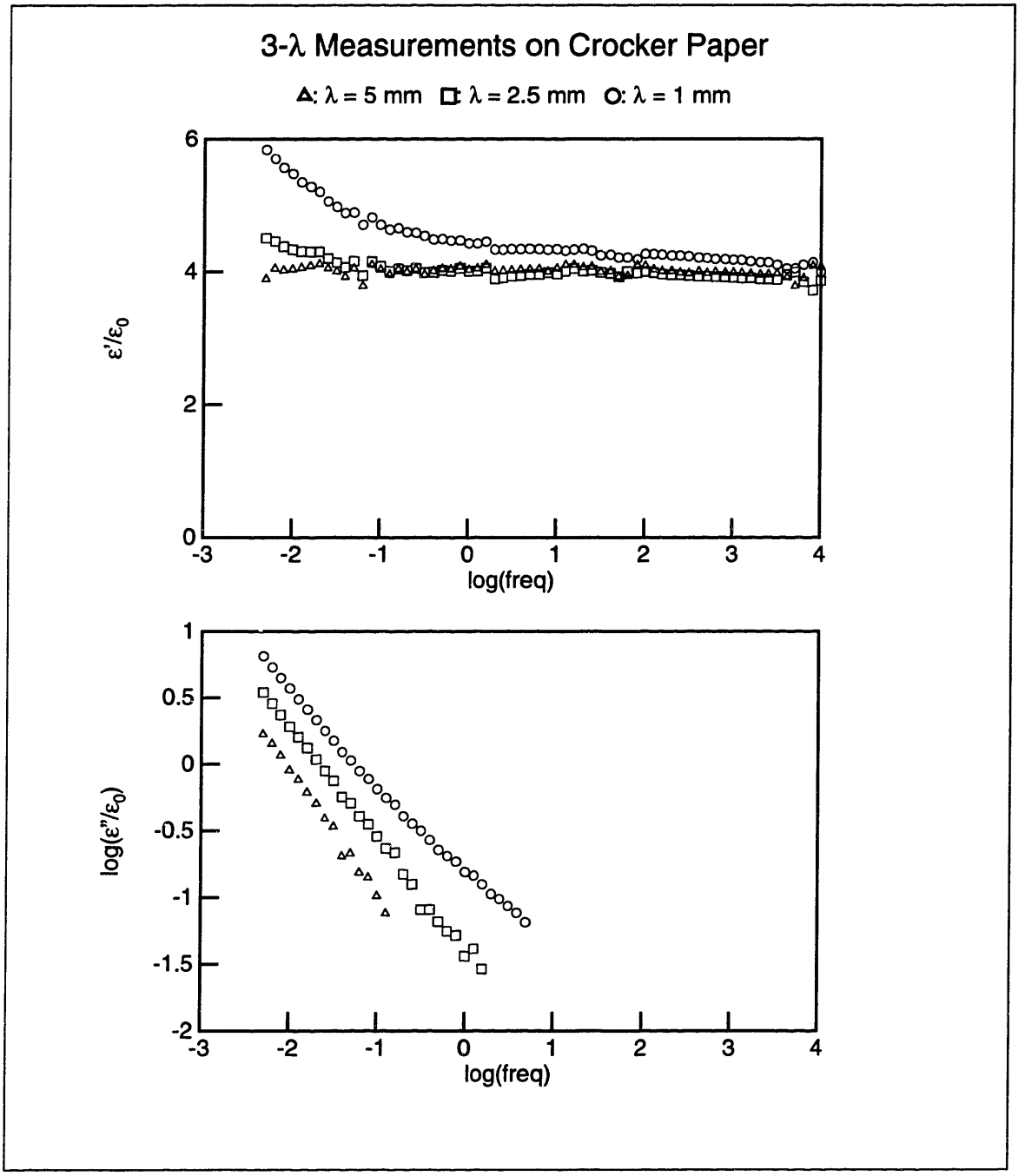

Figure 5-7: Dielectric spectra taken with the three-wavelength sensor on Crocker paper

Time[hours]	ϵ_2	σ_2
1	10.2 ϵ_0	1.41×10^{-11}
6	9.66 ϵ_0	1.29×10^{-11}
12	9.57 ϵ_0	1.23×10^{-11}
57	9.48 ϵ_0	9.49×10^{-12}
86	8.45 ϵ_0	6.24×10^{-12}

Table 5.4: Results of applying the multidimensional parameter estimation algorithm to data at 0.01 Hz taken after the application of vacuum to oil-impregnated Crocker Paper at time $t = 0$. The tabulated values are for the Crocker paper layer closest to the three-wavelength sensor (layer 2).

This estimation was based on a layer 2 thickness of $d_2 = 0.25$ mm. The highly conducting interfacial zone may be much thinner and more highly conducting than these estimates, corresponding to even higher values of the relaxation frequency. This means that at 0.01 Hz the electric fields are shielded from the rest of the paper and the parameter estimation for layers 0 and 1 becomes a victim to a low signal-to-noise ratio.

After that, vacuum was applied to the chamber and continuous frequency scans were taken every hour, in order to monitor the drying process. Figures 5-8, 5-9, and 5-10 show the results from the application of one-dimensional parameter estimation to the data from each individual wavelength. The figures show the dielectric spectra measured by each wavelength independently at five specific times: 1, 6, 12, 57, and 86 hours after the application of vacuum. The dielectric spectra in these figures shift to the left with time, indicating by the decrease in conductivity that moisture is leaving the paper.

The application of the multidimensional search to the same data that yielded the results in Figures 5-8, 5-9, and 5-10 at 0.01 Hz produced the results listed in Table 5.4. The table lists the estimated dielectric properties of the layer closest to the sensor (layer 2), showing the low frequency dispersion by its enhanced permittivity. These results are in agreement with the trends seen in Figures 5-8, 5-9, and 5-10 and provide a quantitative measure of the changes in conductivity associated with the drying. As already discussed, this conducting layer greatly reduced the sensitivity of

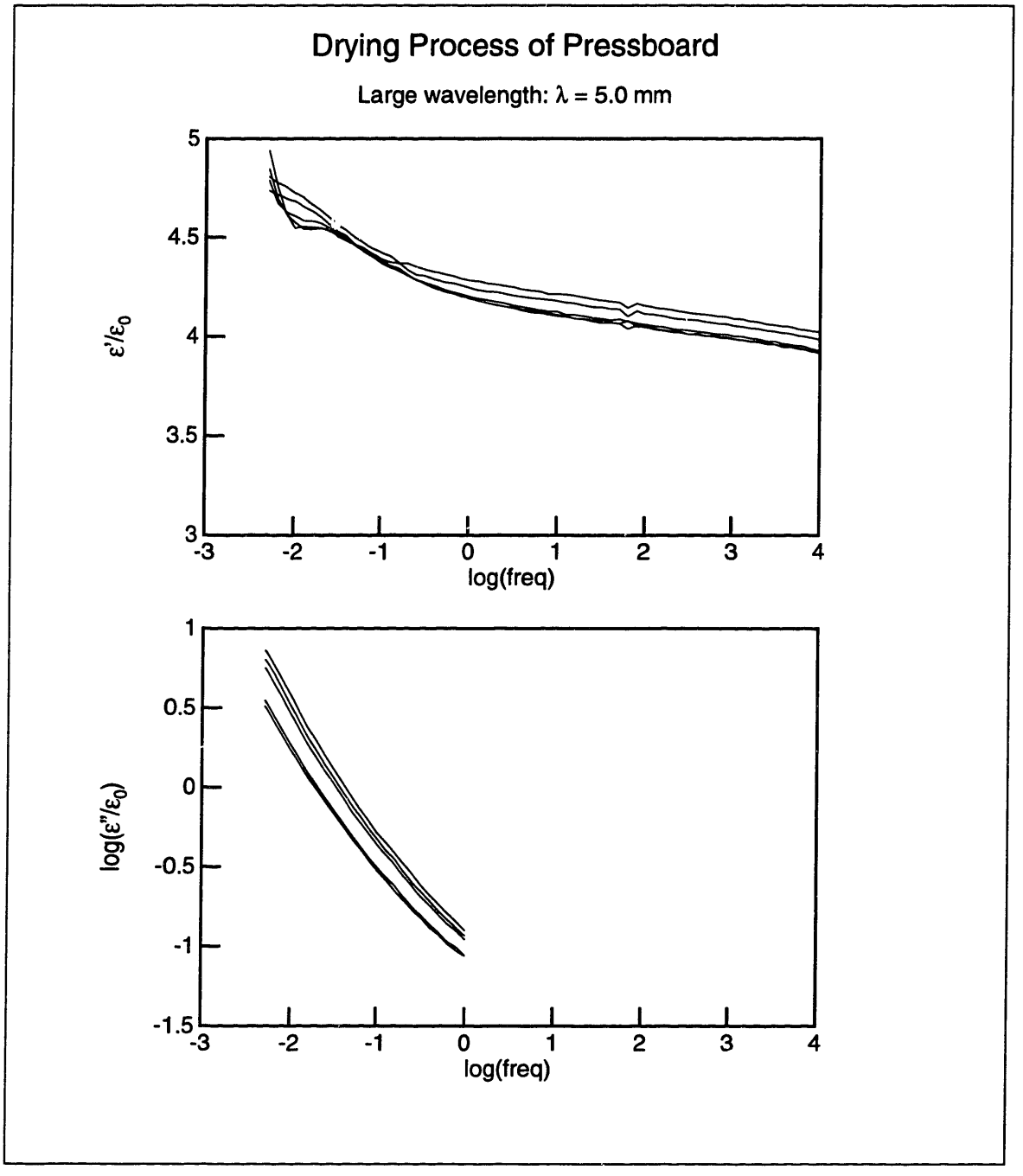

Figure 5-8: Dielectric spectra of oil-impregnated Crocker paper drying under vacuum, taken with the 5.0 mm wavelength of the three-wavelength sensor. The five spectra, in descending order, correspond to frequency scans taken at 1, 6, 12, 57, and 86 hours after the application of the vacuum.

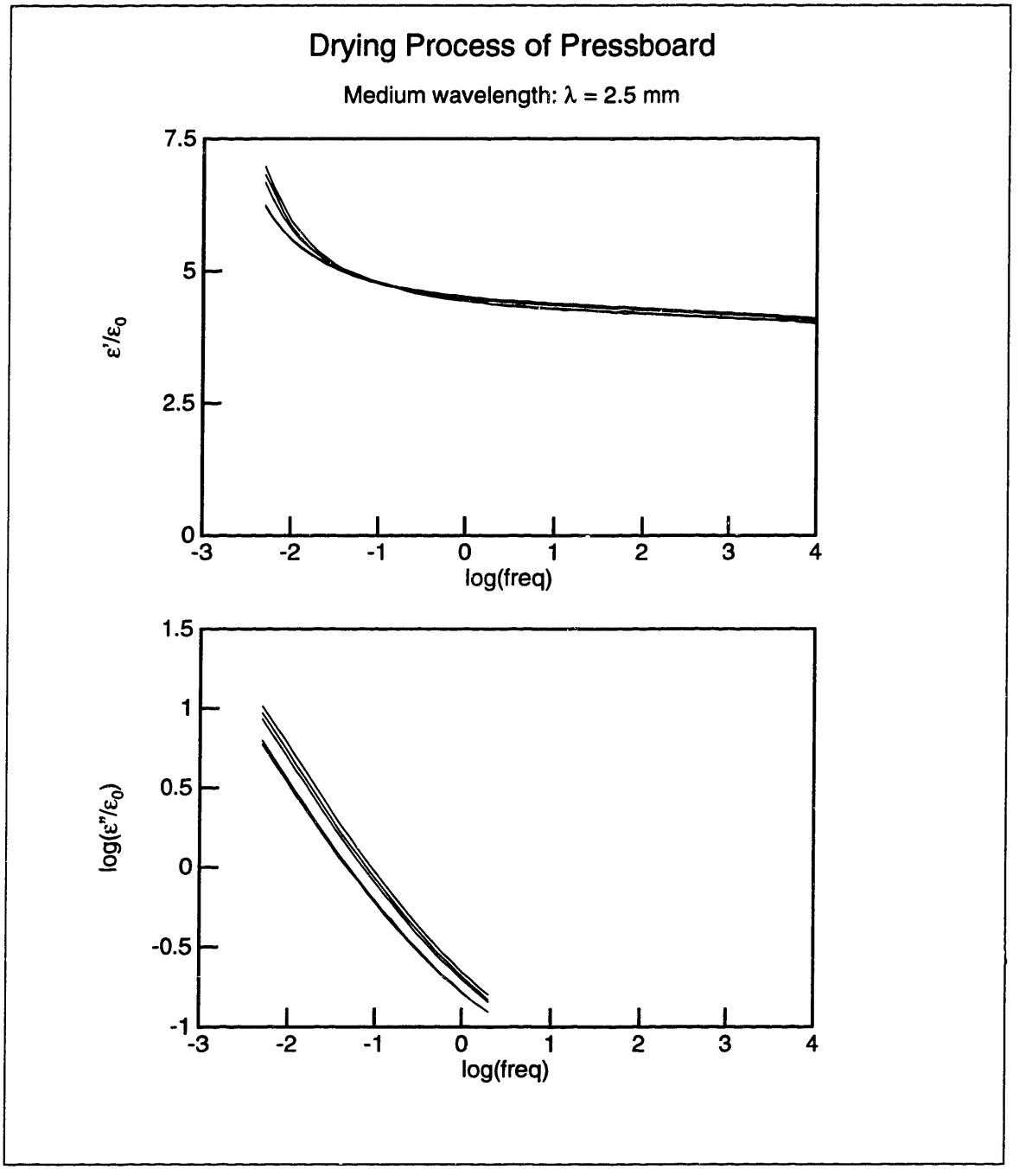

Figure 5-9: Dielectric spectra of oil-impregnated Crocker paper drying under vacuum, taken with the 2.5 mm wavelength of the three-wavelength sensor. The five spectra, in descending order, correspond to frequency scans taken at 1, 6, 12, 57, and 86 hours after the application of the vacuum.

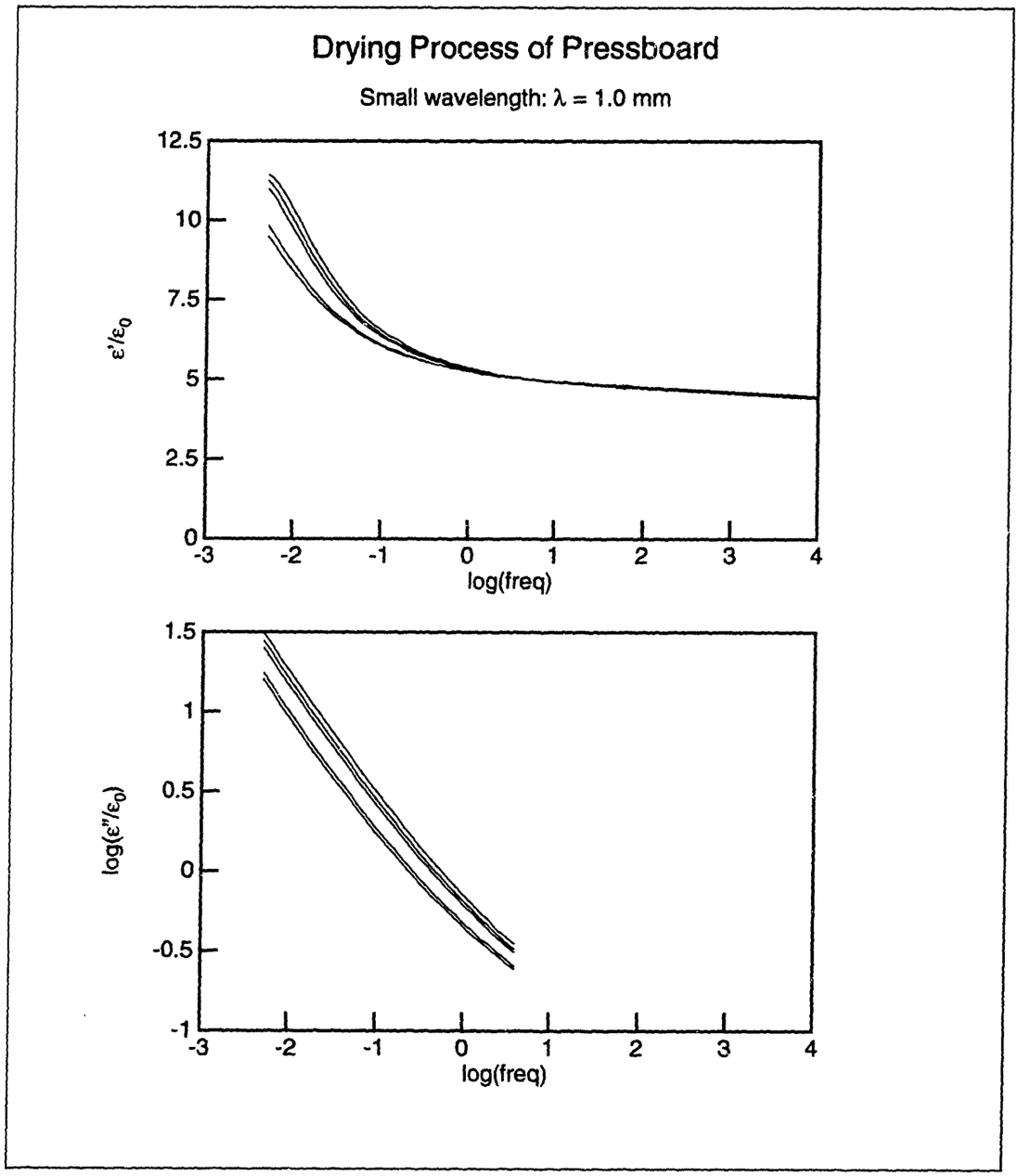

Figure 5-10: Dielectric spectra of oil-impregnated Crocker paper drying under vacuum, taken with the 1.0 mm wavelength of the three-wavelength sensor. The five spectra, in descending order, correspond to frequency scans taken at 1, 6, 12, 57, and 86 hours after the application of the vacuum.

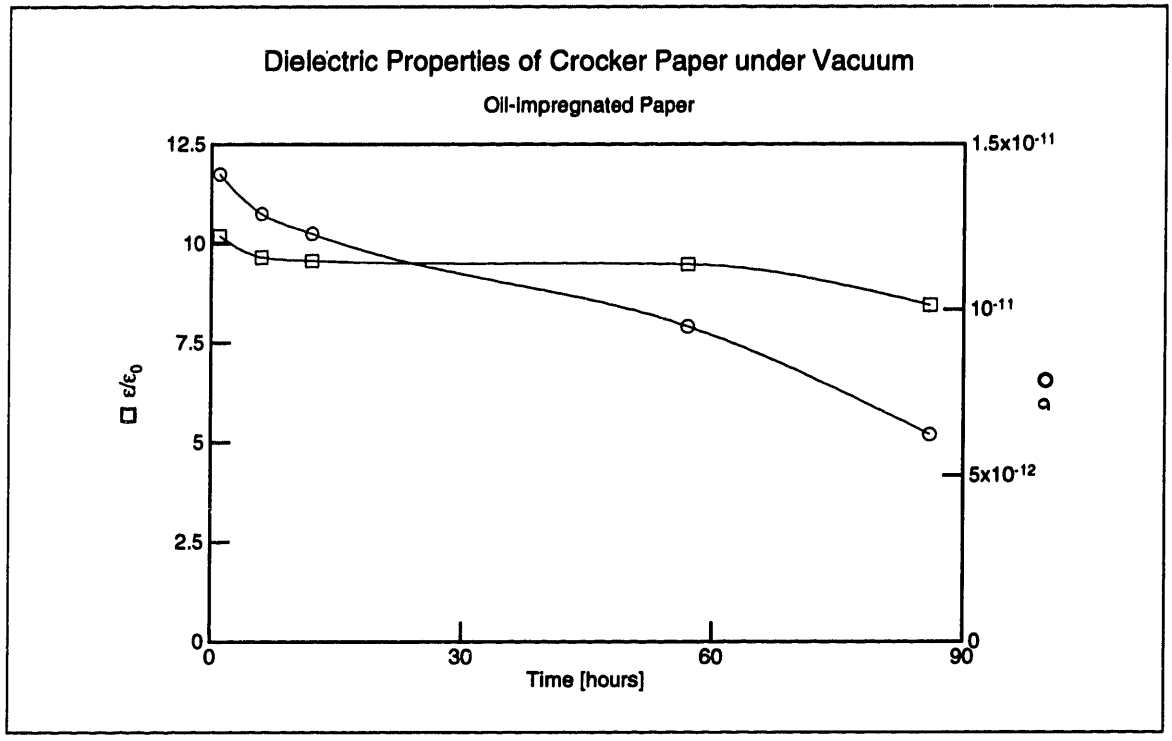

Figure 5-11: Permittivity and conductivity of Crocker paper adjacent to the three-wavelength sensor, calculated by the multidimensional algorithm at 0.01 Hz, as a function of time

this estimation to the properties of the other two layers. The results from Table 5.4 are plotted in Figure 5-11.

Chapter 6

Conclusions

6.1 Universal Spectrum

The importance of the universal dielectric spectrum presented in Chapter 2 is that a relationship may be developed between the dielectric properties and one or more physical properties of a material. While Chapter 2 did present one such universal spectrum, it also established a methodology of obtaining such spectra for a material. The measurement method with the parallel-plate sensor can be used on any material.

While taking dielectrometry measurements with a parallel-plate sensor, it is important that the material is homogeneous. For example, our work showed that for moisture to reach a uniform concentration in the pressboard sample, five days at 50°C were needed.

A method of combining the data from all dielectric spectra was discussed in Section 2.3.2, which allowed all thirty-five measurements to be merged into a single universal curve. This process resulted in a relationship between the manner in which the individual spectra are moved in order to form a single curve and the physical parameters being varied. This relationship could later be used in the opposite direction to obtain moisture content or temperature information from a dielectric measurement.

For oil-impregnated pressboard the dielectric spectra were shifted horizontally with frequency on a logarithmic scale, the shift being a function of the two physical parameters under consideration, temperature and moisture content. An important

conclusion of the analysis of this function was that the effects due to variations in each of these two physical parameters were independent of each other (see equations 2.36 and 2.37).

The limited range of temperature variation, about 13% on an absolute temperature scale, made it difficult to find the exact functional form of the relationship describing the frequency shift due to temperature, although an Arrhenius dependence fit the data quite well. Future work should include testing a wider temperature range.

The relationship describing the frequency shift due to moisture was difficult to establish for a different reason. The calibration moisture measurement with the vaporizer, described in detail in Appendix B, has a wide error margin. Therefore many more data points in Figure 2-17 are necessary in order to establish the functional form with confidence. When it is considered that performing measurements on one pressboard sample takes a week to allow for the conditioning transient to die away, it is clear why only seven data points are present in that figure. In order to find the universal dielectric spectrum of a material together with its accompanying functions, it is necessary to perform many more tests than we were able to do. However, we did establish the procedure that must be followed in order to obtain such universal spectra.

6.2 Parameter Estimation

Parameter estimation is the process of calculating the complex permittivities of the materials above an interdigitated sensor from measured gain-phase data. It is the reverse of finding the gain phase response of a sensor if the material above it is known, often referred to as the forward problem. The forward problem is solved in Section 3.3 for the case of a number of homogeneous layers of constant complex permittivity. In other words, the solution presented in that section is only applicable for cases where the complex permittivity profile is a stair-step function of the spatial variable x. Other spatial profiles can be found as solutions to equation 4.55 in Section 4.5.3.

Parameter estimation is performed numerically, because it is impossible to find

closed-form solutions to the reverse problem. It can be implemented as iterative root-finding techniques or optimization techniques. The simplest case is the one-dimensional search, where the complex permittivity of only one parameter is unknown.

Flexible sensors with different spatial wavelengths may be used to extract information about the spatial profile of the dielectric properties of a material by combining the results of several sensors of different spatial wavelengths.

We have developed three different methods of processing the data from measurements with more than one wavelength: the marching approach, the multidimensional search, and the multidimensional search with an assumed profile function. The first method is simpler and more reliable and it is valid when there is one unknown inhomogeneous layer extending to infinity, but it is not applicable to arbitrary structures. The marching approach and the multidimensional search approximate the profile with a stair-step distribution. The multidimensional search may be done either as a root-finding problem, in which case exact solutions are sought, or as an optimization problem in which case the minimum of an error function is sought. The second option allows for including data from more wavelengths than there are unknowns.

The multidimensional search with an assumed profile function attempts to include in the estimation algorithm some knowledge of the physics of moisture diffusion, by using a smooth function to represent the variation of the dielectric properties of the pressboard across its thickness. It is a root-finding problem where the unknowns are some parameters of this assumed function.

All of these methods need refinement, although we have successfully used them in some applications. One of the major difficulties is that because the forward problem is very non-linear, the multidimensional algorithms may become unstable or otherwise fail to find a root if the initial guess is too far from the solution. This is why it is very important to start with a good first guess to a multidimensional estimation problem, perhaps from applying the one-dimensional algorithm to every individual wavelength first.

Which of these parameter estimation routines is preferred depends on the charac-

teristics of the particular problem.

6.3 Moisture Profiles

We have used the ideas about measuring dielectric profiles, developed in Chapters 3 and 4, on measurements with the three-wavelength sensor on polymers and oil-impregnated paper. One of the obstacles in measuring spatial profiles by probing the material from only one surface is that a highly conducting layer near the surface will limit the electric fields from penetrating into the material and reduce the measurement's sensitivity to the bulk dielectric properties.

Altogether the experiments confirm the feasibility of the method of calculating dielectric profiles. Depending on the application a choice must be made about the spatial wavelengths of the interdigitated sensors, the frequency range, and the most appropriate parameter estimation algorithm, so that the greatest amount of information about the material properties is extracted from the measurements.

Future work should include more diagnostic profile measurements with polymers or other well known materials, selecting their thicknesses in a way that would make the estimation methods sensitive to all layers. Slightly conducting polymers would be a good choice, because this would test the methods under more general conditions than very insulating materials could. More experiments on measuring moisture dynamic processes in pressboard with the three-wavelength sensor are also needed. When confidence is gained in the use of the simpler parameter estimation methods, the method of using an assumed profile function, presented in Section 4.5, should be further studied.

Appendix A

Corollaries of the Kramers-Krönig Relations

Several times in this thesis some properties of the dielectric spectra of materials were used, which follow directly from the Kramers-Krönig relations. In this appendix we present the derivation of these relations and some of their immediate consequences. In the following discussion the following symbols are used: ϵ is the dielectric permittivity; ϵ_∞ is its value for frequencies approaching infinity; σ is the conductivity; σ_0 is the DC conductivity; $\epsilon^* = \epsilon' - j\epsilon''$ is the complex permittivity; ω is the radian frequency. We also define:

$$\epsilon' = \epsilon = \epsilon_0 \chi' + \epsilon_\infty \tag{A.1}$$

$$\epsilon'' = \frac{\sigma}{\omega} = \epsilon_0 \chi'' + \frac{\sigma_0}{\omega} \tag{A.2}$$

$$\chi^* = \chi' - j\chi'' \tag{A.3}$$

The above definition of the complex dielectric susceptibility differs from the usual convention in that it excludes the frequency-independent terms due to ϵ_∞ and σ_0. In our definition, χ^* represents only the *dispersive part* of the complex dielectric susceptibility. This definition is made because, as we prove later, the real and imaginary parts of χ^* are a Hilbert transform pair, and the Hilbert transform of a constant is

zero.

For an ohmic material, whose ϵ and σ are independent of frequency, $\chi^* = 0$, $\epsilon = \epsilon_\infty$, and $\sigma = \sigma_0$. The Kramers-Krönig relations are useful in describing dispersive, i.e. non-ohmic, behavior.

The dispersive part of the *Polarization Density* $\vec{P}$ of a material depends on the electric field intensity $\vec{E}$ in the following way:

$$\vec{P} = \epsilon_0 \chi^* \vec{E} \tag{A.4}$$

Since we are assuming that $\vec{P}$ and $\vec{E}$ always point in the same direction, we'll drop the vector symbols and treat them as scalar quantities. Let us suppose that the applied electric field is an impulse of unity area:

$$E = \delta(t) \tag{A.5}$$

Then the time response of the dispersive polarization density will be described by the impulse response function $h(t)$:

$$P = \epsilon_0 h(t) \tag{A.6}$$

Since χ^* describes the polarization density in the sinusoidal steady state, we can see that the dielectric susceptibility of a material as a function of frequency is the *Fourier transform* of the time-domain impulse response $h(t)$:

$$\chi^*(\omega) = \mathcal{F}\{h(t)\} \tag{A.7}$$

Causality places a rigid constraint on $h(t)$. Since there can be no response before a drive is applied, we know that $h(t) = 0$ for $t < 0$. We can therefore write:

$$h(t) = h(t)u(t) \tag{A.8}$$

where $u(t)$ is the unit step function defined as:

$$u(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases} \tag{A.9}$$

If we take the Fourier transform of both sides and let $f = \omega/(2\pi)$, we obtain:

$$\chi^*(f) = \chi^*(f) * \left[\frac{1}{2}\delta(f) + \frac{1}{j2\pi f}\right] \tag{A.10}$$

$$\chi^*(f) = \chi^*(f) * \frac{1}{j\pi f} \tag{A.11}$$

$$\chi'(f) - j\chi''(f) = \mathbf{P}\int_{-\infty}^{+\infty} [\chi'(\eta) - j\chi''(\eta)] \frac{1}{j\pi(f-\eta)} d\eta \tag{A.12}$$

where the asterisk $*$ indicates the operation of *convolution.* If we now equate the real and imaginary parts of the left and right sides of equation A.12, we obtain:

$$\chi'(f) = -\frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty} \frac{\chi''(\eta)}{f-\eta} d\eta \tag{A.13}$$

$$\chi''(f) = \frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty} \frac{\chi'(\eta)}{f-\eta} d\eta \tag{A.14}$$

Equation A.14 is in the form of a *Hilbert transform* and χ' and χ'' are said to be a *Hilbert transform pair* [10, pp. 479]

The Kramers-Krönig relations simply rewrite these equations to obtain [1, sec. 2.8]:

$$\chi'(\omega) = \frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty} \frac{\chi''(x)}{x-\omega} dx \tag{A.15}$$

$$\chi''(\omega) = -\frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty} \frac{\chi'(x)}{x-\omega} dx \tag{A.16}$$

The **P** in front of the integral symbol indicates that this is a *Cauchy principal value* integral, i.e. the imaginary contributions to the integral, which come from passing through the pole at $x = \omega$, and which cancel [8, sec. 10.15], are ignored. If we take into account that χ' is an even function and χ'' is odd, which is a direct consequence

of $h(t)$ being real [10, pp. 379], we may rewrite these relations in the following form:

$$\begin{aligned}
\chi'(\omega) &= \frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty}\frac{\chi''(x)}{x-\omega}dx = \frac{1}{\pi}\left[\mathbf{P}\int_{-\infty}^{0}\frac{\chi''(x)}{x-\omega}dx + \mathbf{P}\int_{0}^{+\infty}\frac{\chi''(x)}{x-\omega}dx\right] \\
&= -\frac{1}{\pi}\left[\mathbf{P}\int_{0}^{-\infty}\frac{\chi''(x)}{x-\omega}dx - \mathbf{P}\int_{0}^{+\infty}\frac{\chi''(x)}{x-\omega}dx\right] \\
&= \frac{1}{\pi}\left[\mathbf{P}\int_{0}^{\infty}\frac{\chi''(-x)}{-x-\omega}dx + \mathbf{P}\int_{0}^{+\infty}\frac{\chi''(x)}{x-\omega}dx\right] \\
&= \frac{1}{\pi}\mathbf{P}\int_{0}^{\infty}\chi''(x)\left(\frac{1}{x+\omega}+\frac{1}{x-\omega}\right)dx \\
&= \frac{2}{\pi}\mathbf{P}\int_{0}^{+\infty}\frac{x\chi''(x)}{x^2-\omega^2}dx \qquad \text{(A.17)} \\
\chi''(\omega) &= -\frac{1}{\pi}\mathbf{P}\int_{-\infty}^{+\infty}\frac{\chi'(x)}{x-\omega}dx = -\frac{1}{\pi}\left[\mathbf{P}\int_{-\infty}^{0}\frac{\chi'(x)}{x-\omega}dx + \mathbf{P}\int_{0}^{+\infty}\frac{\chi'(x)}{x-\omega}dx\right] \\
&= \frac{1}{\pi}\left[\mathbf{P}\int_{0}^{-\infty}\frac{\chi'(x)}{x-\omega}dx - \mathbf{P}\int_{0}^{+\infty}\frac{\chi'(x)}{x-\omega}dx\right] \\
&= -\frac{1}{\pi}\left[\mathbf{P}\int_{0}^{\infty}\frac{\chi'(-x)}{-x-\omega}dx + \mathbf{P}\int_{0}^{+\infty}\frac{\chi'(x)}{x-\omega}dx\right] \\
&= -\frac{1}{\pi}\mathbf{P}\int_{0}^{\infty}\chi'(x)\left(-\frac{1}{x+\omega}+\frac{1}{x-\omega}\right)dx \\
&= -\frac{2\omega}{\pi}\mathbf{P}\int_{0}^{+\infty}\frac{\chi'(x)}{x^2-\omega^2}dx \qquad \text{(A.18)}
\end{aligned}$$

A.1 Parallel Shifts

The Kramers-Krönig relations require that when plotted on a log-log scale, a shift in the plot of χ' must correspond to a shift in the plot of χ'' by the same amount both horizontally and vertically. From equations A.15 and A.16 it is clear that a constant multiplying χ' would change χ'' by the same amount, as the constant can be pulled out of the integral. A vertical shift on a logarithmic scale corresponds to multiplication by a constant.

Now examine the horizontal shifts, which correspond to multiplying the frequency variable by a constant. For example, what χ_1' corresponds to $\chi_1''(\omega) \equiv \chi''(k\omega)$? If we make the substitution $x' = kx$, we may write:

$$\begin{aligned}\chi_1'(\omega) &= \frac{2}{\pi}\mathbf{P}\int_0^\infty \frac{x\chi_1''(x)}{x^2-\omega^2}dx = \frac{2}{\pi}\mathbf{P}\int_0^\infty \frac{x\chi''(kx)}{x^2-\omega^2}dx \\ &= \frac{2}{\pi}\mathbf{P}\int_0^\infty \frac{(x'/k)\chi''(x')}{(x'/k)^2-\omega^2}\cdot\frac{dx'}{k} = \frac{2}{\pi}\mathbf{P}\int_0^\infty \frac{x'\chi''(x')}{x'^2-(\omega k)^2}dx' = \chi'(k\omega) \quad \text{(A.19)}\end{aligned}$$

Similarly

$$\begin{aligned}\chi_1''(\omega) &= \frac{2\omega}{\pi}\mathbf{P}\int_0^\infty \frac{\chi_1'(x)}{x^2-\omega^2}dx = \frac{2\omega}{\pi}\mathbf{P}\int_0^\infty \frac{\chi'(kx)}{x^2-\omega^2}dx \\ &= \frac{2\omega}{\pi}\mathbf{P}\int_0^\infty \frac{(\chi'(x')}{(x'/k)^2-\omega^2}\cdot\frac{dx'}{k} = \frac{2\omega}{\pi}\mathbf{P}\int_0^\infty \frac{k\chi'(x')}{x'^2-(\omega k)^2}dx' = \chi''(k\omega) \quad \text{(A.20)}\end{aligned}$$

This proves that horizontal shifts in χ' and χ'' must also be of the same magnitude.

A.2 Same Slopes

On a log-log plot, to the right of the rightmost peak of χ'', both χ' and χ'' decrease with the same slope. The slope of decrease to the right of a loss peak is negative and ranges between -1 and 0 [1]. It can be proved that if for $\omega \to 0$, $\chi'' \propto \omega^m$, $-\infty < m < 1$, then as $\omega \to \infty$, $\chi'' \propto \omega^n$ and $\chi' \propto \omega^n$ with the same slope n. In this appendix an example of a typical dielectric spectrum is illustrated, which shows this property.

Suppose χ'' of a dielectric consists of a single loss peak at ω_p and follows the power law

$$\chi''(\omega) = \frac{K}{\sqrt{\frac{\omega}{\omega_p}} + \sqrt{\frac{\omega_p}{\omega}}} = \frac{K\sqrt{\omega\omega_p}}{\omega+\omega_p} \quad \text{(A.21)}$$

i.e. the magnitude of the slope of decrease on either side of the peak is 1/2. Paper has been observed to have a dielectric spectrum which can be expressed in a form similar to equation A.21, and this is why we chose this formulation in this example

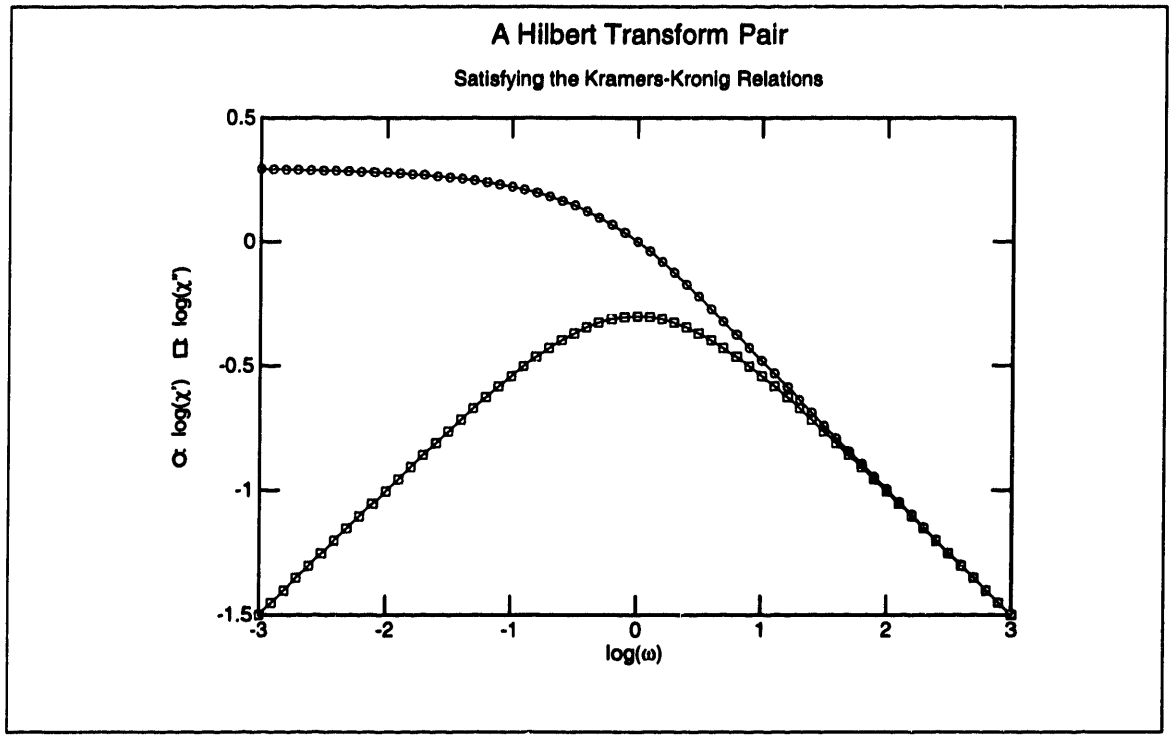

Figure A-1: This is a plot of equations A.21 and A.22 for $\omega_p = 1$ and $K = 1$. As $\omega \to \infty$ both curves decrease with a slope of $-1/2$.

[11, Fig. 5.14 and 5.15]. Let us then perform the integration using equation A.17:

$$\begin{aligned}
\chi'(\omega) &= \frac{2}{\pi}\mathbf{P}\int_0^\infty \frac{x\,dx}{x^2-\omega^2}\cdot\frac{K\sqrt{x}\sqrt{\omega_p}}{x+\omega_p} \\
&= \frac{K\sqrt{\omega_p}}{\pi(\omega^2-\omega_p^2)}\mathbf{P}\int_0^\infty \left[(\omega-\omega_p)\frac{\sqrt{x}}{x-\omega} - (\omega+\omega_p)\frac{\sqrt{x}}{x+\omega} + 2\omega_p\frac{\sqrt{x}}{x+\omega_p}\right]dx \\
&= \frac{K\sqrt{\omega_p}}{\pi(\omega^2-\omega_p^2)}\left[(\omega-\omega_p)\sqrt{\omega}\ln\left|\frac{-\omega-x+2\sqrt{\omega x}}{x-\omega}\right| + \right. \\
&\qquad \left. + 2\sqrt{\omega}(\omega+\omega_p)\tan^{-1}\sqrt{\frac{x}{\omega}} - 4\omega_p\sqrt{\omega_p}\tan^{-1}\sqrt{\frac{x}{\omega_p}}\right]_0^\infty \\
&= \frac{K\sqrt{\omega_p}}{\pi(\omega^2-\omega_p^2)}\left[\frac{\pi}{2}2\sqrt{\omega}(\omega+\omega_p) - \frac{\pi}{2}4\omega_p\sqrt{\omega_p}\right] \\
&= \frac{K\sqrt{\omega\omega_p}(\omega+\omega_p)-2\omega_p^2}{\omega^2-\omega_p^2} \qquad \text{(A.22)}
\end{aligned}$$

For $\omega \rightarrow \infty$ both χ'' and χ' are proportional to $1/\sqrt{\omega}$. A log-log plot of these functions is shown in Figure A-1.

Appendix B

Water Vaporizer Moisture Measurements

In order to measure the moisture content of pressboard sample, it is placed in the oven of a Mitsubishi VA-05 Vaporizer, where it is subjected to an elevated temperature, usually between 100°C and 200°C. Dry nitrogen gas is flowed through the oven and bubbled through the titration cell of a Mitsubishi CA-05 Moisture Meter, where water is trapped and its quantity is measured by Karl-Fisher titration.

In an attempt to determine the optimal temperature of the oven, we took a series of measurements on pieces of the same oil-impregnated pressboard sample. The results are displayed in Figures B-1, B-2, and B-3.

The moisture meter determines the total quantity of moisture introduced in the titration cell by monitoring the speed of titration and integrating it over the duration of the measurement. There is a background level of titration, due to small amounts of moisture entering the system through leakages and within the nitrogen gas. The measurement terminates when the speed of titration drops to a level slightly higher than the background level.

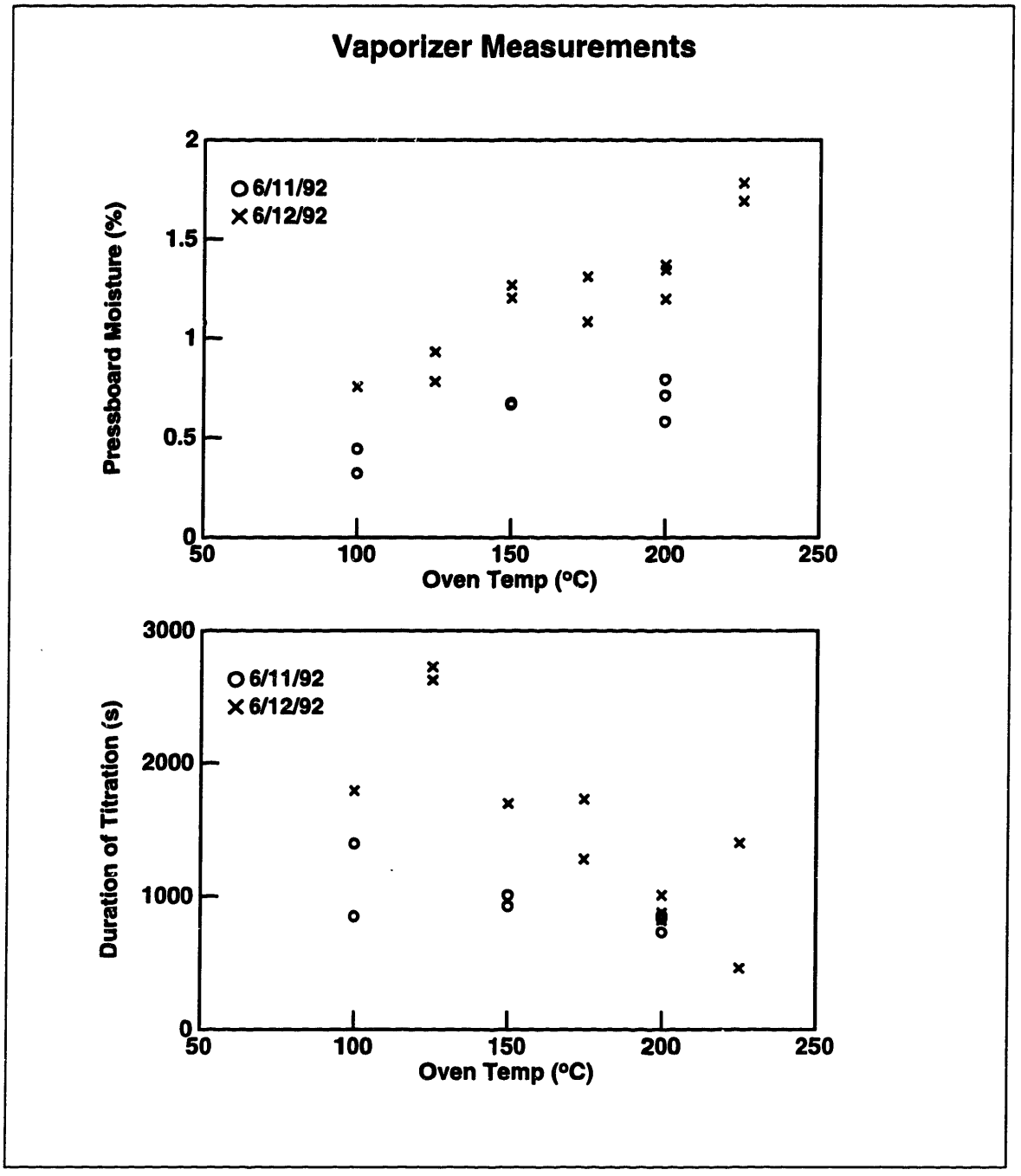

Figure B-1: These plots show how the measured pressboard moisture and the duration of titration depend on the temperature of the oven. They were taken with 1mm thick HIVAL pressboard. Two sets of data were taken on subsequent days. The increase in the readings on the second day suggests that the samples, which were stored in air, absorbed some moisture during the course of the day. In subsequent measurements samples were stored under oil and no appreciable change in the moisture content was observed.

B.1 Effect of Sample Thickness on Moisture Measurement

While testing for the optimal temperature, discussed in the next section, we performed measurements on samples of the same moisture content, but different thicknesses. In order to ensure that the samples did indeed have the same moisture content, we impregnated a piece of 1 mm thick pressboard by subjecting it to vacuum at 70°C (see Appendix C) for an unusually long period of time (more than 48 hours), thus ensuring that equilibrium with the vacuum was reached. From this piece of pressboard we then created samples of various thicknesses by peeling off a different number of plies.

The results of this set of measurements are shown in Figure B-2. This figure clearly shows that the thinner samples produced higher readings. One of the 1 mm samples, shown with an asterisk in Figure B-2, was split in many thin layers before being placed in the oven. Its measured moisture content was much higher than that of the other 1 mm samples and comparable to the thinnest samples. This indicated that the difference in the readings of the samples of different thicknesses was not due to a difference in their moisture contents, but to the fact that the rate of diffusion of water was so low for the thick samples, that it became comparable to the background titration level and was not properly registered.

We concluded that as a standard procedure all pressboard samples should be split into many plies before they are placed into the vaporizer oven. We have followed this procedure in all measurements described in Chapter 2.

B.2 Optimal Temperature

We expected that at lower temperatures (100–140°C) the method would underestimate the amount of moisture, as not all of the moisture would diffuse out of the sample by the end of the measurement and the rate of water liberation might be comparable to the background level. At high temperatures (>200°C) cellulose be-

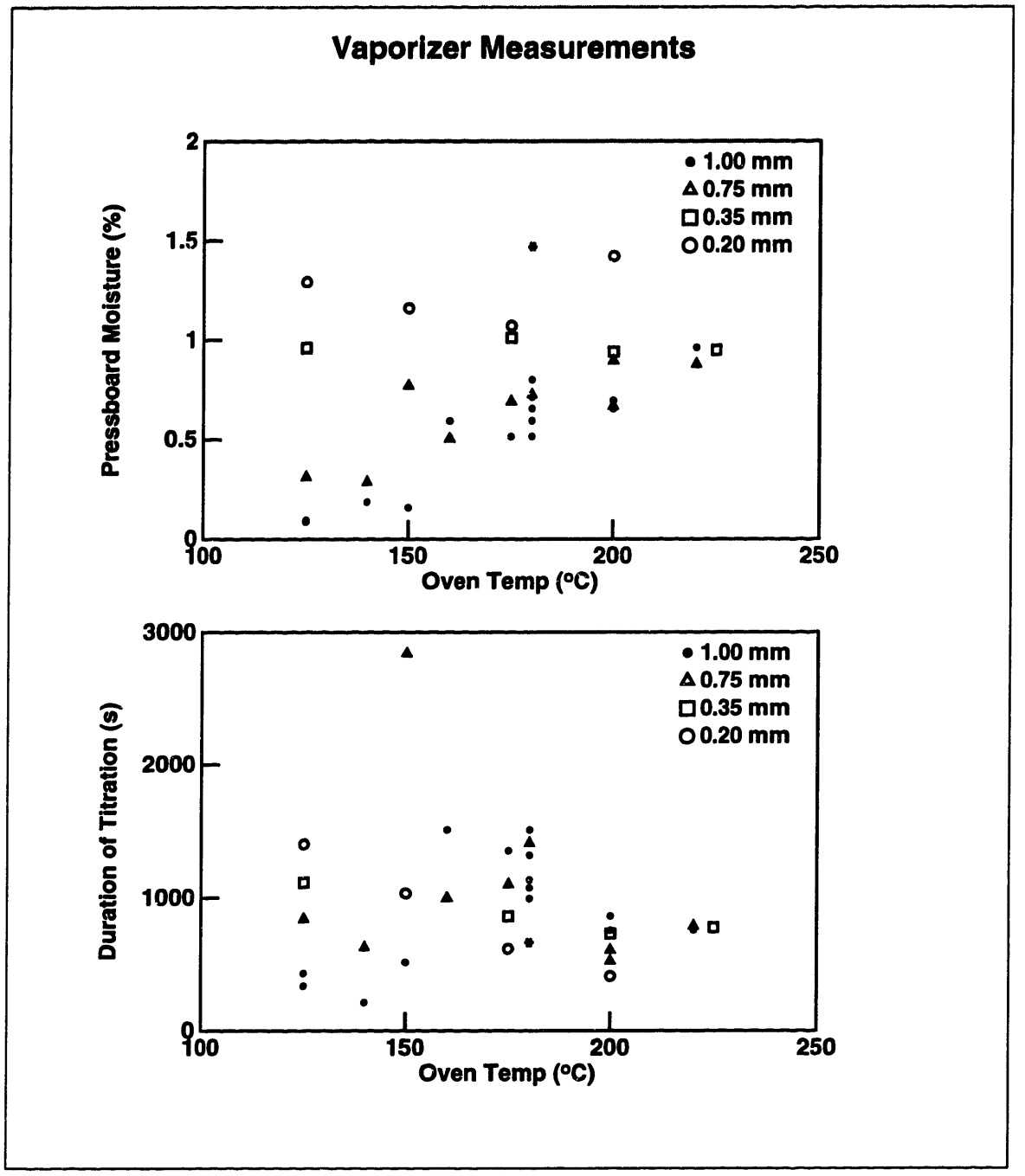

Figure B-2: These plots show how the measured pressboard moisture and the duration of titration depend on the temperature of the oven. Samples of different thickness of the same material with the same moisture content were used. The asterisk represents a data point taken with a 1 mm thick sample, which was split in thin layers, all placed together in the oven. The thinner samples read higher moisture content values. It is clear that the measurement is greatly affected by the thickness of the samples.

gins to disintegrate, liberating bonded water, thus causing an overestimate in the measurement. We therefore tried to determine the range of oven temperatures when neither of these extreme phenomena occur.

Once we had established that for reliable moisture measurements of pressboard the samples had to be thin, we conducted another set of experiments with 82 μm thick Crocker paper, which is another similar insulating cellulose material. The results are shown in Figure B-3. We concluded that if the oven temperature is between 100°C and 200°C, it does not affect the value of the moisture measured. The duration of titration decreased with temperature in that range. Therefore temperatures at the higher end of the operating range (180–200°C) were the preferred choice, as they lead to lower titration times. The uncertainty of the measurement, calculated from the data in Figure B-3, was 17%.

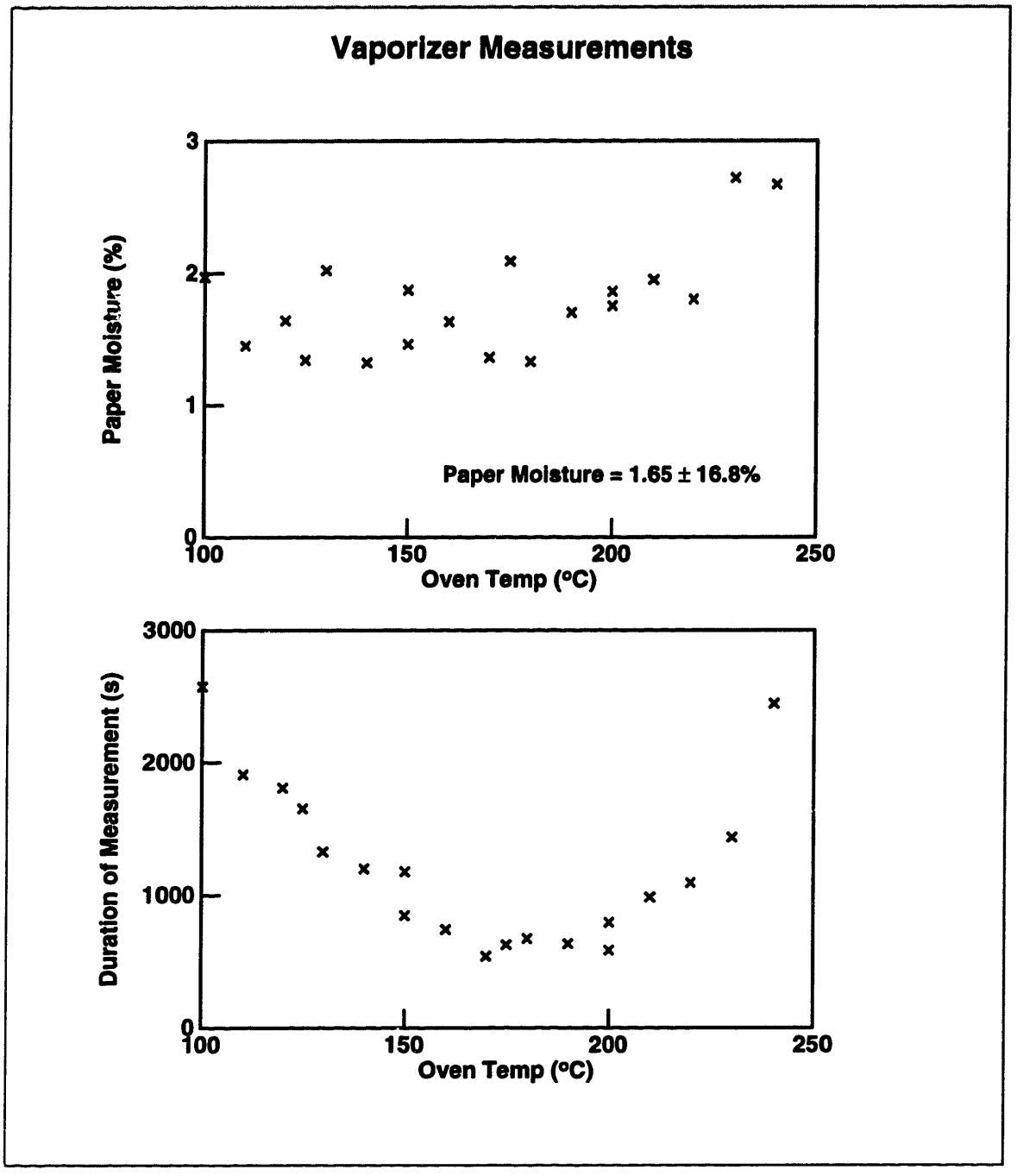

Figure B-3: In order to avoid the limitations of measuring thick samples, thin oil-impregnated paper (Crocker, 82 μm) is used to investigate how the temperature of the oven affects the moisture measurement. The duration of titration consistently decreases with increasing temperature, indicating higher diffusion rates, up to a temperature above which paper disintegration begins. The latter observation is confirmed by the higher values of moisture content obtained for temperatures above 200°C.

Appendix C

Procedures for Oil-Impregnation of Pressboard and Paper

This appendix describes the procedures we have followed for impregnating our pressboard and paper samples with transformer oil. We have made an attempt to simulate the impregnation procedure that is followed commercially for the manufacturing of high-power transformers.

There are two stages to the process. The first stage involves drying of the oil-free pressboard under vacuum. This is done at an elevated temperature to facilitate the diffusion of moisture as it leaves the pressboard. The second stage entails immersing the dry pressboard in transformer oil, which has been heated to speed up its absorption. The pressboard is kept under vacuum right up to the time it is immersed in the oil.

Figure C-1 shows the structure of the oil-impregnation facility that we have used. The two interconnected chambers are made out of stainless steel. All openings are vacuum-sealed. Valves 3 and 4 are three-way valves that have three settings: closed; center-left; and center-right. The chamber to the right of the figure is used to store the transformer oil and may be filled/emptied via the two inlets at the top and the bottom by appropriately setting valves 3 and 4.

Since only the oil-impregnation chamber itself is subjected to vacuum, only its vacuum probe is connected to a meter. The temperature probe is used by a tem-

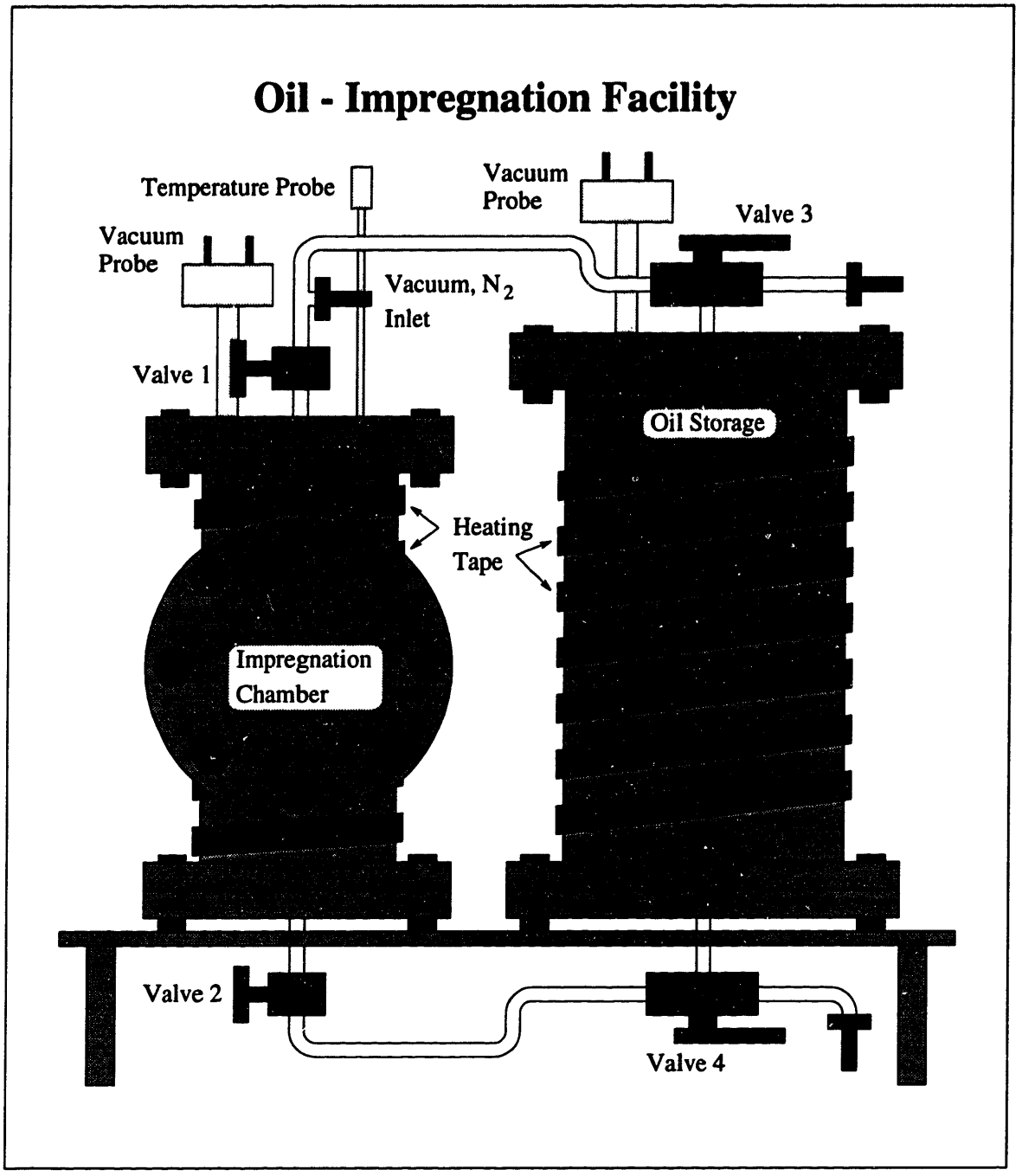

Figure C-1: This is the structure of the apparatus used in the process of impregnation of pressboard and paper samples with transformer oil. Its operation is described in the text.

perature controller, which has the dual function of displaying the temperature and controlling the heating tape so as to maintain a constant pre-specified temperature.

Here is the sequence of events associated with impregnating a pressboard or a paper sample: At first valves 1 and 2 are open, valve 3 connects the oil chamber with the inlet to the air, and valve 4 is in the closed position. The samples are placed in the empty impregnation chamber through the removable front flange, which is then tightly closed. A vacuum pump is connected to the vacuum inlet. The heater is turned on before the pump is, because heat transfer to the sample is much better when the chamber is full of air rather than vacuum. After the desired temperature is reached, typically 70°C, the vacuum pump is turned on. We let the pressure decrease to the lowest possible value, which is about 100 mTorr, (20 mTorr on good days), and keep the sample under vacuum for the desired length of time, typically overnight. At this point valve 1 is closed and the pump is turned off and disconnected. It is important to follow this sequence, because otherwise oil may enter the vacuum pump. Now valve 4 is set to connect the oil storage chamber to the oil-impregnation chamber, thereby letting the room air pressure push the oil in. The sample is kept under oil, still at a high temperature, for about two hours. After that the oil is forced out of the impregnation chamber by connecting a nitrogen gas source to the vacuum inlet and opening valve 1. When the chamber has been emptied, valve 4 is closed, which returns the unit to its original state. The pressboard samples may then be extracted through the same flange in the front.

A concern has been expressed that since the oil is continuously exposed to the ambient air, it will saturate with water and introduce a large amount of moisture into the pressboard as the oil itself enters the pressboard. This is not a concern, because the solubility of water in oil is so small compared to its solubility in cellulose, that the amount of water present in the oil which enters the pressboard is truly negligible compared to the amount of moisture still present in the cellulose at the end of the drying stage.

The parameters of the impregnation process presented in this appendix are only approximate. The exact conditions are appropriately listed in the text. The materi-

als used were EHV-Weidmann HIVAL pressboard, 82 μm Crocker paper, and Shell Diala A transformer oil.

Appendix D

Controller

In this appendix we present information about the operation of the *controller*, a microprocessor-based data acquisition system. It was developed by Mr. David Otten and full instructions for its operation may be obtained from the MIT LEES staff.

The controller is capable of two types of data acquisition: the Data Logger, which we have not used, and Gain-Phase Measurements (GPM). The controller communicates to a computer via an RS-232 line, which is used both to send appropriate commands and to receive data. Table D.1 presents a summary of commands recognized by the controller. Both upper case and lower case letters are acceptable.

The controller has a total of four *channels* for its GPM operation, i.e. it is capable of processing four independent inputs. It provides all four channels with a *driven* AC voltage of complex amplitude $\widehat{V}_d$. It performs a *frequency scan*, i.e. measurements at all frequencies in a specified range. The maximal range of frequencies at which the controller is capable of performing GPM measurements is from $10^{-2.3}$ Hz (≈ 0.005 Hz) to 10^4 Hz. Two consecutive measurements are at frequencies 0.1 apart on a logarithmic scale, corresponding to a ratio of $10^{0.1} \approx 1.259$. At every frequency, the controller waits for a certain number (3–7) of cycles to complete so as to ensure that sinusoidal steady state is reached, at which point it records the magnitude and phase angle of the ratio between the input voltage and the driven voltage, in decibels

and degrees respectively:

$$\text{Gain} = 20\log\left|\frac{\hat{V}_{in}}{\hat{V}_d}\right| \tag{D.1}$$

$$\text{Phase} = \frac{180}{\pi}\angle\left(\frac{\hat{V}_{in}}{\hat{V}_d}\right) \tag{D.2}$$

The input impedance of the controller is not high enough for applications in which very insulating materials are studied. Therefore buffering of the input signal is needed and that is provided by the interface box, described in Appendix E. In addition to gain and phase data, another piece of information recorded by the controller is the offset voltage, which is the DC component of the input voltage. This DC buildup is due to charge accumulated on the input capacitance from the input current of the operational amplifier. This phenomenon is further discussed in the appendix describing the interface box. The controller may be offset adjusted.

Only one controller is at this time able to be connected to the interface box of the three-wavelength sensor described in Chapter 3 and it is currently connected to the computer "LEES-OMEGA-K" via the port name **ttyaf**. The controllers were designed to accommodate two channels per interface box connector, but this controller had some extra wiring added so that three channels could be connected via the same cable.

As data is collected by the controller, it is stored in its internal memory until this memory buffer is explicitly cleared. The controller may be set to begin new measurements periodically, which is extremely useful if the experiment requires monitoring a process as it evolves with time. However, in this mode the controller will quickly run out of memory, after which it will stop recording data. If a single channel is enabled, e.g. when the parallel-plate sensor of Section 2.1 is used, the controller will run out of memory after thirteen full frequency scans. If, on the other hand, three channels are enabled, e.g. for three-wavelength sensor measurements, the memory will be enough to store data for only five full frequency scans.

To avoid this problem, a program was developed which automatically stores the

All commands are made up of an opening bracket [, two letters, optional parameters for some commands, and a closing bracket]. For those commands which allow parameters, the current status of the parameters will be displayed if they are omitted. If the parameters are included, they will be updated. All the parameters or none of them must be used.

Command	Description
[FP,parameters]	read/set fixed parameters
[GP,parameters]	read/set GPM parameters
[LP,parameters]	read/set data logger parameters
[TG]	trigger gain phase meter
[TL]	trigger data logger
[GD]	read GPM data
[LD]	read logger data
[CS]	check status
[AM]	abort any data logger and GPM measurements in progress
[CM]	clear memory buffer
[DT,parameters]	read/set date and time
[CP,parameters]	read/set communication parameters
[ME]	master auto-trigger flag enable
[MD]	master auto-trigger flag disable
[VE]	software version number

Fixed parameters - not supported, not supported, channel 1 delay, channel 2 delay, channel 3 delay, channel 4 delay

GPM parameters - starting frequency, ending frequency, excitation level, channel 1 enable, channel 2 enable, channel 3 enable, channel 4 enable, diagnostic enable, auto trigger enable

Data Logger Parameters - channel 1 gain, channel 2 gain, channel 3 gain, channel 4 gain, channel 5 gain, channel 6 gain, channel 7 gain, auto zero enable, auto trigger enable

Date and Time parameters - year, month, day, hour, minute, second

Communication parameters - RS-232 baud rate, telephone number, data storage interval, data dump interval, call enable

GPM header - channel number, temperature, year, month, day, hour, minute, second

GPM data - frequency, magnitude, phase, offset, gain

Data Logger data - channel 1 data, channel 2 data, channel 3 data, channel 4 data, channel 5 data, channel 6 data, channel 7 data, year, month, day, hour, minute, second

Status - master auto-trigger flag, not supported, Data Logger measurement, not supported, GPM measurement, not supported, not supported

Table D.1: Summary of Controller Commands

data from every scan into a file and clears the memory. It is called **tw.c** and is presented in Section G.2. The other way of communicating with the controller is the program **kermit**, available at most UNIX systems. If the automatic program **tw.c** is used, it is recommended that a low baud rate is set, e.g. 1200, so that transmission errors are reduced to a minimum. It is usually all right to use a baud rate of 9600 when using **kermit**, because any communication problems would be easily detected visually.

Since the gain measured by the controller may vary greatly in the range of frequencies of interest, the controller may have to switch between different modes of pre-amplification. It has been noted that such transitions between modes may result in erroneous data at a specific frequency, manifested as 'kinks' in the otherwise smooth curves relating the gain and phase response of a system to the frequency. These events are merely experimental artifacts and no physical significance should be attributed to them.

Appendix E

Interface Boxes

The interface boxes' main function is to buffer a voltage signal, before it is processed by the controller and to raise its input impedance. The buffering is accomplished in two ways: a unity-gain-connected operational amplifier provides a very high input impedance at the sensitive node; and the buffered signal thus obtained is used to guard the sensing electrode and wiring. The guarding of the electrode is accomplished via special guard electrodes, present both in the parallel-plate and the three-wavelength sensors. The connecting cable is triaxial, with the middle connected to the guard potential. Since the sensing and the guard electrodes are always at the same potential (for frequencies less than the dominant pole of the operational amplifier), any parallel parasitic impedance is effectively multiplied by the gain of the amplifier, thus making its effects negligible.

E.1 Parallel-Plate Sensor Interface Box

The schematic diagram of this box's circuit is shown in Figure E-1. The input of the box is loaded with a parallel RC pair, whose values are precisely known. It is crucial for the interpretation of data that this load impedance be known. Therefore the values chosen for these elements are such that the parasitics associated with the operational amplifier are negligible. The relay is used to discharge the load capacitor every time a measurement is completed, in order to prevent saturation. The transistor

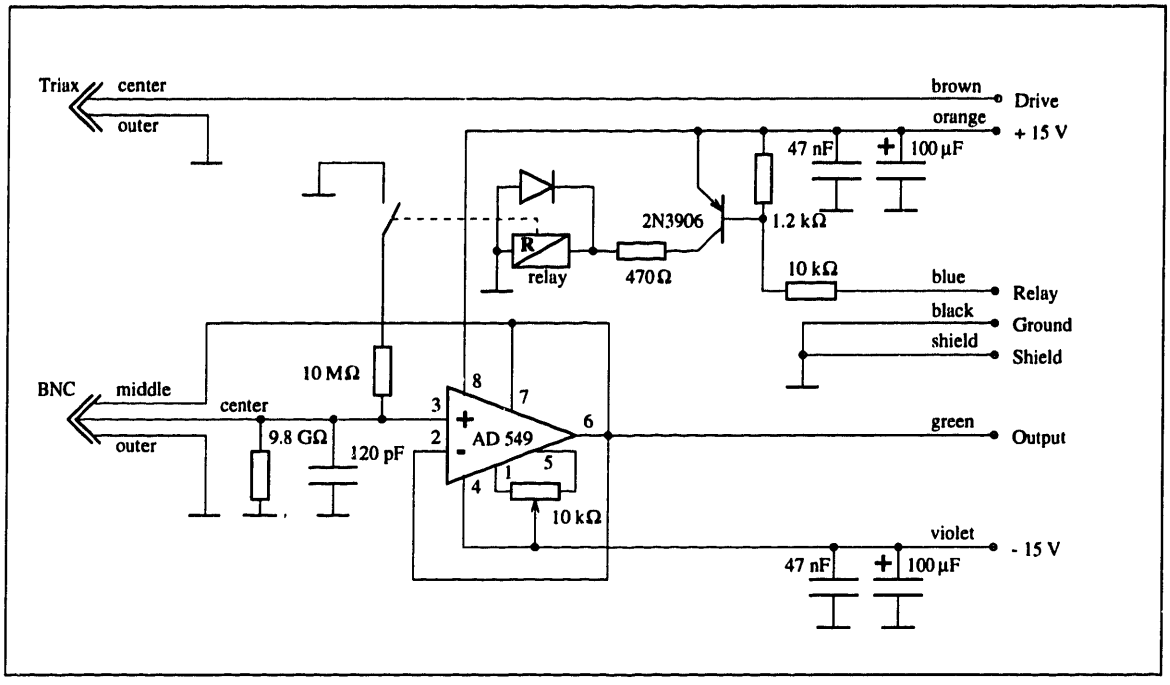

Figure E-1: Interface box circuit diagram

is used to drive the relay.

The value of the load impedance, as well as the overall performance of the interface box, may be tested by connecting a known reference lumped-element parallel RC pair between the driven potential and the sensing input and then processing the data with the program **testrc.c**, described in Section G.4. The output of the program are values for the estimated load impedance at a full range of frequencies. If the box is operating properly, these values should be close to those in Figure E-1 and independent of the frequency of excitation. Figure E-2 shows the output of the program **testrc.c** when applied to a test frequency scan, where the load cell was replaced by a parallel RC-pair of values of $R_T = 48.9$ GΩ and $C_T = 120$ pF. The bottom plot shows the estimated values of C_L and R_L, also listed in Table E.1.

E.2 Three-Wavelength Sensor Interface Box

The circuit of this interface box is essentially identical to that shown in Figure E-1, but repeated three times, one circuit for each channel. Instead of BNC connectors

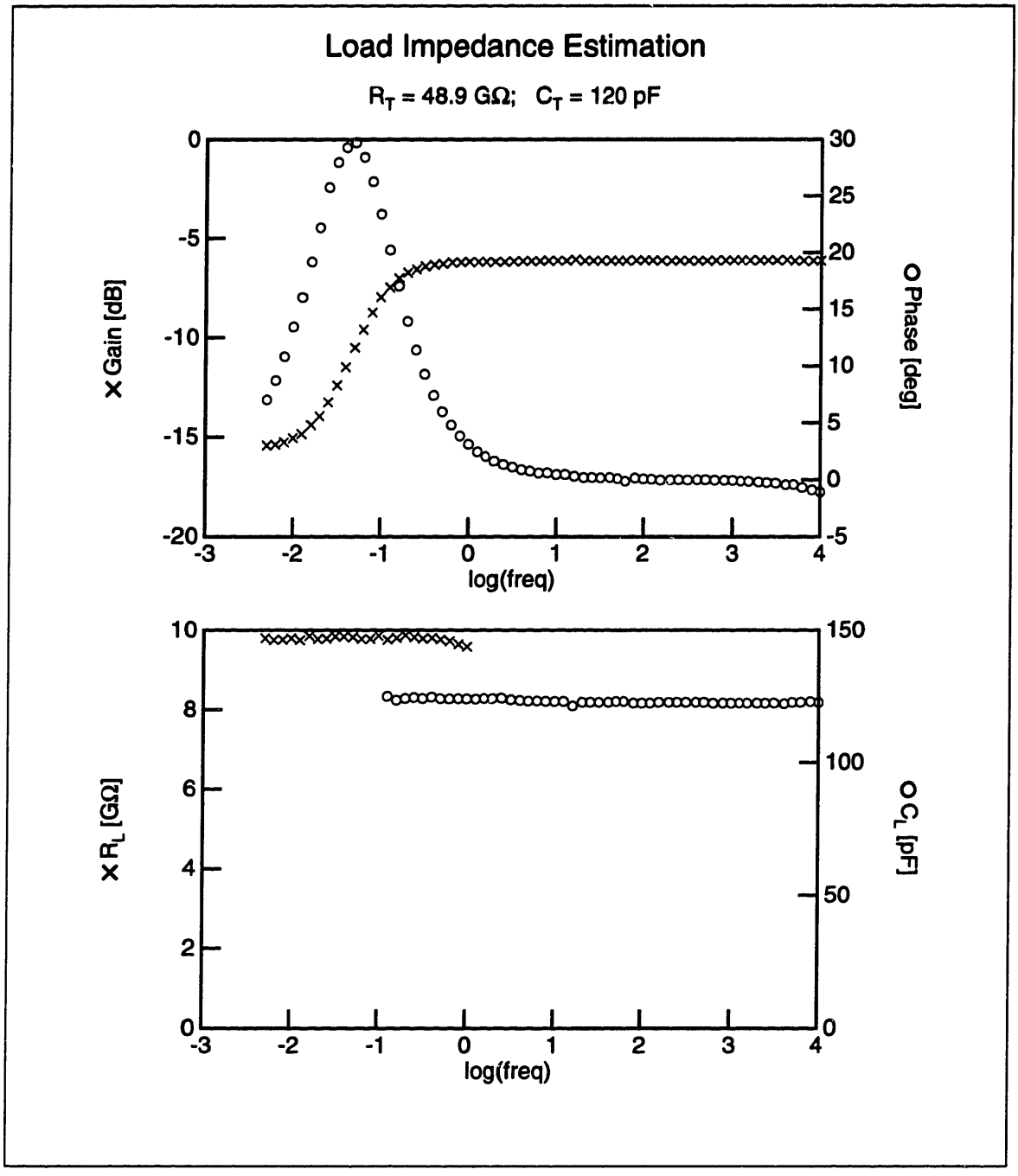

Figure E-2: Results from measurements of the load impedance of the parallel-plate sensor's interface box

Load Impedance	Parallel-Plate Sensor	Three-Wavelength Sensor		
		5.0 mm	2.5 mm	1.0 mm
C [pF]	123	6.47	37.6	226
R [GΩ]	9.78	∞	∞	∞

Table E.1: Interface Box Load Impedances

this box features a special connector, which fits the leads of the three-wavelength sensor (see Figure 3-1). It also differs in the load impedances, as listed in Table E.1. As discussed in Section 2.1.3, no sensitivity is lost when no load resistance is present. The value of 6.47 pF in Table E.1 is the parasitic input capacitance of the operational amplifier. Capacitors were added at the input of the other two channels with values that would make the gain of the three-wavelength sensor in air close to -40 **dB**. The input resistance of these amplifiers is extremely high ($10^{12}\Omega$ or higher) and has no influence on the measurements even under worst case conditions (see Section 3.4).

Appendix F

Mathematical Examples

Sets of Equations with Real Unknowns That Require Extra Degrees of Freedom

We present an example that would illustrate the principles discussed in Section 4.1.2. The idea is that if a set of real unknowns always appear in real clusters in every equation of the set of complex equations, then either there is no solution, or more equations would be needed for a unique solution.

Consider the following set of equations:

$$\begin{cases} r_1 r_2 z + (2+j) = 0 \\ (r_1 + r_2) z - (6+3j) = 0 \end{cases} \tag{F.1}$$

According to the general rules from Section 4.1.2, a total of four degrees of freedom are needed, two for the two real variables r_1 and r_2 and two for the complex variable z. The two complex equations should then suffice to determine the unknowns uniquely. However, eliminating z from the second equation by substituting the first into it yields

$$\begin{cases} r_1 r_2 z + (2+j) = 0 \\ \frac{r_1 + r_2}{r_1 r_2} - 3 = 0 \end{cases} \tag{F.2}$$

which clearly shows that another equation is necessary to find unique solutions for r_1 and r_2. Information has been lost in the requirement that $(6+3j)/(-2-j) = -3$ be

a real number. If it had not been so, the set of equations would have had no solution. For the set of equations F.1 the real clusters, are $(r_1 r_2)$ and $(r_1 + r_2)$.

As a counterexample, the set of equations

$$\begin{cases} r_1 + r_2 z + (3+j) = 0 \\ (r_1 + r_2) z - (6+3j) = 0 \end{cases} \tag{F.3}$$

can easily be solved to yield $r_1 = -1$, $r_2 = 0.25$, and $z = -8 - 4j$.

Exact Determination of a Function of a Known Form with a Limited Number of Degrees of Freedom

This is the idea that motivates the method of parameter estimation with an assumed profile function, discussed in Section 4.5. Suppose that we know that a function $f(x)$ has a parabolic dependence on x of the form

$$f(x,p) = apx^2 + bx + c\sqrt{p} \tag{F.4}$$

and that we are able to measure the result of some operation on f for any value of the parameter p, e.g. a simple integration

$$F(p) = \int_{x_1}^{x_2} f(x,p)dx \tag{F.5}$$

where x_1 and x_2 are constants. The results of the measurements give us three values for $F(p)$, say F_1, F_2, and F_3, for $p = p_1, p_2, p_3$ respectively. This results in the set of equations:

$$\begin{cases} F(p_1) = F_1 \\ F(p_2) = F_2 \\ F(p_3) = F_3 \end{cases} \tag{F.6}$$

which would let us solve for a, b, and c, resulting in an exact determination of the function f. Only three degrees of freedom were necessary in this case to yield an error-free solution. A stair-step approximation seems clearly an inadequate option, if

the exact form can be obtained. This is what has been gained by introducing *a priori* knowledge about the functional form of the solution in the problem.

To make the process even clearer, let us give the parameters numerical values. So, here is the problem from the beginning: We have a function of x, which when integrated between $x_1 = 0$ and $x_2 = 1$ for three different values of some parameter $p = 0, 1, 4$ yield the values -1, $4/3$, and $13/3$ respectively. If we assume the functional form of equation F.4, we may integrate it to obtain the following set of equations:

$$\begin{cases} b = -2 \\ 2a + 3b + 6c = 8 \\ 8a + 3b + 12c = 26 \end{cases} \tag{F.7}$$

which can be solved to give $a = 1$, $b = -2$, and $c = 2$.

Appendix G

Program Listings for Data Processing Software

G.1 Description

All programs listed in this appendix are summarized in Table G.1. The rest of this section provides somewhat more information.

G.1.1 Data Acquisition

testrc.c This program reads a file generated by the controller box and outputs to **stdout** plotting commands to plot the estimated load resistance and load capacitance, when the corresponding test values are known. Used for interface box diagnostics.

tw.c This program takes data at regular intervals and stores it in files with a name given as an argument with subsequent numbers appended to it. It directly accesses the controller via the port **ttyaf**.

Name	Description
testrc.c	Calculates load impedances
tw.c	Records data for the controller
clean.c	"Cleans" controller data
divide.c	Splits data file according to channel number
do.c	Manipulates data files
domerge.c	Merges data files
nothing.c	Sinks data
only.c	Eliminates noise from data files
rev.c	Reverses a file
separate.c	Separates data files according to scan
inv.c	Calculates ϵ' and ϵ'' from controller data
lstsq.c	Performs least-squares fit
out2e.c	Interprets output of **parestsx**
rcinv.c	Calculates R_T and C_T from controller data
extrapolate.c	Extrapolates data by power law
fit.c	Fits data to power law
fith.c	Calculates logarithmic frequency shifts
fitm.c	Fits data to power law
kk12.c	$\epsilon' \rightarrow \epsilon''$ via Kramers-Krönig
kk21.c	$\epsilon'' \rightarrow \epsilon'$ via Kramers-Krönig
maximum.c	Finds maximum in an array of data
power_fit.c	Fits data to power law
reverse.c	Reverses an array of data
sen12.c	Sensitivities of $Z_T \rightarrow Z_L$ process
sen21.c	Sensitivities of $Z_L \rightarrow Z_T$ process
ecomp.c	Plots and compares ϵ^* data
eplot.c	Plots ϵ^* data
eplotx.c	Plots ϵ^* data
eplot3.c	Plots ϵ^* data for three-wavelength sensor
eplot3x.c	Plots ϵ^* data for three-wavelength sensor

Table G.1: Summary of data processing software

G.1.2 Low-Level Data Processing

clean.c This program reads a file generated by the controller box and outputs to **stdout** the frequency, gain, and phase data, suitable for input files to the parameter estimation routines. If the phase is positive it is set to zero.

divide.c It is used on files produced by the controller box if more than one channel is active. The file must be **separated** first. The output files have names with an extension according to the number of the channel. If an option '-b' is specified, a special file naming convention is used: channel 1 starts with 'p', channel 3 starts with 'f'. Source file in this case must start with 'b'.

do.c This is a useful little program which manipulates pairs of data. It takes its input from **stdin** and writes its output to **stdout**. The command line should indicate what to do with the numbers. Key letters are:

a (add) followed by a number, add the number

s (scale) followed by a number, scale the number

l (log10) take its log10

p (pow10) take its antilog10

n (nothing) leave number as is.

Some examples:

```
do l a -1.0 < infile > outfile
do n l < infile > outfile
do a 3.14 s 2.75 < infile > outfile
extrapolate < infile | kk | do l n > outfile
```

domerge.c This program takes as an argument a template data file which consists of pairs of numbers on individual lines. It outputs to **stdout** in the same format, but with the first number replaced by a number read from **stdin**.

nothing.c This program serves as a data sink for useless output. If the standard output of a program is not needed and it is a waste to send it to a file or to a screen, it can be piped into this program.

only.c It cuts off parts of a data file. **lb** data points from the end are discarded. On the front side, all data points with x-coordinate greater than **rb** are also discarded. In this sense **lb** and **rb** *are not* equivalent: one is an integer and the other is a float. It is assumed that the x-coordinates are in *descending* order. Usage: `only <file> <lb> <rb>`

rev.c This program reverses the lines in a file.

separate.c It takes a file which contains output of the controller box and separates it into individual files which contain one set of measurements each. The original file name must end on 'x' and the new files have the 'x' substituted with consecutive letters of the alphabet. In order for this to work, the original [**gd**] command must be on a new line. An optional argument specifies the maximum number of files to be written. Usage: `separate <file> [max]`

G.1.3 High-Level Data Processing

inv.c It takes a file produced by the controller box and outputs two files containing data for ϵ' and ϵ'' with extensions **.e1** and **.e2** respectively. An optional second argument specifies a setup file which contains information about the interface box. The default is `/u/yanko/.invsetup`.

lstsq.c This is a function which does a least-squares fit of a line to a set of data points. It is used by other routines.

out2e.c It takes as an input an output file of **parestsx**[1] and produces two output data files containing the data for ϵ' and ϵ'' with extensions **.e1** and **.e2** respectively,

[1] Developed by M. Zaretsky. See [3].

replacing the **.out** extension of the input file.

rcinv.c It takes a file produced by the controller box and outputs two files containing data for R and C with extensions **.rr** and **.cc** respectively. A file **.invsetup** with information on the interface box must exist in the home directory **/u/yanko**.

G.1.4 Data Interpretation

extrapolate.c This program takes a set of data representing the dependence of ϵ'' on frequency and extrapolates the data assuming values for the slope on either side of the peak to be of equal magnitude and opposite sign. The data should then be easy to integrate using the Kramers-Krönig relations to obtain values for ϵ'. The data is read from **stdin** and output to **stdout**. The value of the slope is printed out to **stderr**.

fit.c This program fits a power-law curve to a set of data supplied from **stdin**. It has an optional argument specifying the value of m. The program outputs the values of k and f_p to **stderr** and pairs of data points along the fitted curve to **stdout**. The function of the curve being fitted is:

$$y = \frac{k}{\left(\frac{x}{f_p}\right)^m + \left(\frac{f_p}{x}\right)^m}$$

fith.c This program takes a reference file and a test file and computes by how much the latter would have to be shifted in frequency to produce a least squares sum. The input files have extensions **.r2** and are the reversed versions of **.e2** files. They need to be reversed with the program **rev.c**.

fitm.c A version of **fit.c** which also attempts to fit a value for m.

kk12.c This program uses the Kramers-Krönig relations to calculate ϵ'' from ϵ'. Data is read from **stdin** and output to **stdout**. The integration contains a singularity about the point of frequency being calculated. Therefore the integration is done in

three parts: a small interval *df* on either side of *f* is neglected, because the area around the singularity cancels out. For this cancellation to be valid, the integrand must stay constant. Therefore *df* is taken to be one step of frequency to the left. The integration is then carried out independently to the left and to the right of this interval. Since *df* would be greater on the right side of the singularity, due to the logarithmic step in frequency, there is a need to add on the thin sliver equal to the difference between the two *df*'s on either side. The numerical calculation uses the trapezoidal rule.

kk21.c Exactly the same as **kk12.c**, but it calculates ϵ' from ϵ''.

maximum.c This function is used by other routines to find the largest number in an array of numbers.

power_fit.c This function is used by other routines to perform a non-linear least-squares-fit to a data set according to the following formula:

$$y = \frac{k}{\left(\frac{x}{f_p}\right)^m + \left(\frac{f_p}{x}\right)^m}$$

reverse.c This function reverses the order of elements in an array.

sen12.c This is a program which plots the relative sensitivity of the inversion process from Z_T to Z_L (see Section 2.1.3 to variations in the phase data. Its output is the file **out**, which is ready to be **plop**ped.

sen21.c Same as **sen21.c**, except the inversion process of Z_L to Z_T is considered. This program was used to generate the data in Figure 2-5.

G.1.5 Plotting

ecomp.c This program when piped into **plop** plots the **.e1** and **.e2** files of the given argument. It compares two sets of files by plotting them together.

eplot.c, eplotx.c These programs plot the **.e1** and **.e2** files of the given argument. The second program has predetermined axis scales.

eplot3.c, eplot3x.c These programs when piped into plop plot the **.e1** and **.e2** files of the three wavelengths of the three wavelength sensor. The files

name.1.e1 name.2.e1 name.3.e1

name.1.e2 name.2.e2 name.3.e2

must exist in that directory. The second program has predetermined axis scales.

raw.c, raw3.c These programs plot the raw gain-phase-offset data produced by the controller for the parallel-plate and the three-wavelength sensors respectively.

G.2 Data Acquisition

test.c

```
# include       <stdio.h>
# include       <math.h>
# define        PI      3.14159265      /* PI */
# define        EOL     '\n'            /* end of line definition */
# define        MAXPTS 500
/* This program reads a file generated by the controller box and outputs
   to stdout plotting commands to plot the estimated load resistance and
   load capacitance, when the corresponding test values are known.

        Yanko Sheiretov         9/11/92                                 */

main(argc, argv)
int     argc;
char    *argv[];

{
        char    junk;
        FILE    *fpses;
        double  ch, r1, c1, f[MAXPTS], g[MAXPTS], p[MAXPTS], o, dt, dg, dp, dx;
        int n, i;

        if (argc != 2) {
                fprintf(stderr, "Usage:  \ttestrc <file> | plop\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {
            fprintf(stderr, "Error:  cannot open %s\n",argv[1]);
            exit(1);
            }

        fprintf(stderr, "Enter Test Resistance [GOhm]:  ");
```

```
        scanf("%lf", &r1);
        fprintf(stderr, "Enter Test Capacitance [pF]: ");
        scanf("%lf", &c1);

        while((junk=fgetc(fpses))!=EOL);
        for(n=0;(junk=fgetc(fpses))!=EOF;++n) {
                junk=fgetc(fpses);
                while((junk!=',') && (junk!=']')) {                    40
                        junk=fgetc(fpses);
                        }
                if (junk==',') {
                fscanf(fpses,"%lf,%lf,%lf,%lf,%lf",&ch,f+n,g+n,p+n,&o);}
                while((junk=fgetc(fpses))!=EOL);
                }
        fclose(fpses);
        n--;
        printf("window top\nlabel left \"\\cross Gain [dB]\"\n");
        printf("title top \"Load Impedance Estimation\"\n");           50
        printf("label top \"R_1 = %8.2fG\\Omega; C_1 = %8.2fpF\"\n",r1,c1);
        printf("text over 0.1 right 0.3 size 0.07 \"file:  %s\"\n", argv[1]);
        printf("label right \"\\circle Phase [deg]\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("plot cross green marker 0.08\n");
        for(i=0;i<n;i++) printf("%f\t%f\n",f[i],g[i]);
        printf("plot circles green marker 0.08 use right\n");
        for(i=0;i<n;i++) printf("%f\t%f\n",f[i],p[i]);
        printf("window bottom\nlabel left \"\\cross R_2 [G\\Omega]\"\n");
printf("label right \"\\circle C_2 [pF]\"\nlabel bottom \"log(freq)\"\n");   60
        printf("plot cross green marker 0.08\n");
        for(i=0;i<n;i++) {
                dt = 2.*PI*pow(10.,f[i])*r1*c1*1.e-3;
                dg = pow(10., g[i]/20.);
                dp = p[i]*PI/180.;
                dx = cos(dp)-dg+dt*sin(dp);
                if (dx != 0.) printf("%f\t%f\n", f[i], r1*dg/dx);
                }
```

```
        printf("plot circles green marker 0.08 use right\n");
        for(i=0;i<n;i++) {                                                    70
                dt = 2.*PI*pow(10.,f[i])*r1*c1*1.e-3;
                dg = pow(10., g[i]/20.);
                dp = p[i]*PI/180.;
                printf("%f\t%f\n", f[i], c1*(cos(dp)-dg-sin(dp)/dt)/dg); }

        fprintf(stderr,"Done ...  %d data points.\n",n);
        fflush(stdout);
}
```

```
/*****************************************************************/
/*                                                               */
/*                          tw.c                                 */
/*                                                               */
/*                        11/12/90                               */
/*                                                               */
/*       Modified by Yanko Sheiretov  6/17/93, 3/31/94           */
/*                                                               */
/*  This program takes data at regular intervals and stores      */
/*  it in files with a name given as an argument with            */
/*  subsequent numbers appended to it.                           */
/*                                                               */
/*    Usage: tw <nameroot>                                       */
/*                                                               */
/*****************************************************************/

# include     <stdio.h>       /* define standard I/O routines    */
# include     <sgtty.h>       /* define stty and gtty calls      */
# include     <fcntl.h>       /* define access modes             */

# define      BAUD            B1200
# define      LINE            "/dev/ttyaf"
# define      BUSY            -1
# define      MAXSCANS        99

char          ss[80];
FILE          *fpi,*fpo,*fopen();

main(argc,argv)
int     argc;
char    *argv[];
{
        int     port, ext;
        struct  sgttyb tty;
```

```
        char    filename[80];

        if (argc != 2) {
                fprintf(stderr, "usage:  \ttw <nameroot>\n");
                exit(); }
/*                                                                      40
        set port baud rate, no echo, and raw mode
*/
        if ((port = open(LINE,O_RDWR)) == BUSY) {
                printf("Error:  line busy\n");
                exit(1);
                }
        gtty (port,&tty);
        tty.sg_flags &= (~ECHO);
        tty.sg_flags |= (RAW);
        tty.sg_ispeed = tty.sg_ospeed = BAUD;                           50
        stty(port,&tty);

        fpo = fdopen(port,"r");
        fpi = fdopen(port,"w");
        command("[MD]");
        command("[AM]");
        command("[CM]");
        command("[GP,4.0,-2.3,1.0,E,E,E,D,D,D]");

        for(ext=1;ext<=MAXSCANS;ext++) {                                60
                sprintf(filename, "%s%d%d", argv[1], ext/10, ext%10);
                printf("Acquiring %s ...  ", filename);
                acquire(filename);
                printf("done.\n"); }

        fclose(fpi);
        fclose(fpo);
        close(port);
        }
                                                                        70
```

```
acquire(datafile)
char    datafile[];
{
        char    status;
        FILE    *fpd;

        command("[TG]");
        do {
                system("sleep 60");                                          80
                command("[CS]");
                sscanf(ss,"[CS,%*c,%*c,%*c,%*c,%c,%*c,%*c]",&status);
                printf("status = %c\n",status);
                }
        while (status != 'D');
        fpd = fopen(datafile,"w");
        fprintf(fpi,"[GD]");
        do {
                fgets(ss,80,fpo);
                fprintf(fpd,ss);                                             90
                }
        while (ss[3] != ']');
        command("[CM]");
        fclose(fpd);
        }

command(s)
char    s[];
{
        printf("%s", s);                                                    100
        fprintf(fpi,s);
        fgets(ss,80,fpo);
        printf("\t%s", ss);
}
```

G.3 Low-Level Data Processing

clean.c

```
# include       <stdio.h>
# define        EOL     '\n'            /* end of line definition */
# define        MAXPTS 500

/* This program reads a file generated by the controller box and outputs
   to stdout the frequency, gain, and phase data. If the phase is positive
   it is set to zero.
        Yanko Sheiretov         12/1/92         12/16/93                */

main(argc, argv)                                                        10
int     argc;
char    *argv[];

{
        char    junk;
        FILE    *fpses;
        double ch, f[MAXPTS], g[MAXPTS], p[MAXPTS], o;
        int n, i;

        if (argc != 2) {                                                20
                fprintf(stderr, "Usage:  \tclean <file> > <outfile>\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {
            fprintf(stderr, "Error:  cannot open %s\n",argv[1]);
            exit(1);
            }

        while((junk=fgetc(fpses)!=','));                                30
        while((junk=fgetc(fpses))!=EOL);
        for(n=0;(junk=fgetc(fpses))!=EOF;++n) {
```

```
            junk=fgetc(fpses);
            while((junk!=',') && (junk!=']')) {
                junk=fgetc(fpses);
                }
            if (junk==',') {
            fscanf(fpses,"%lf,%lf,%lf,%lf,%lf",&ch,f+n,g+n,p+n,&o);}
            while((junk=fgetc(fpses))!=EOL);
            if (p[n] > 0.0) p[n] = 0.0; /* set positive phase to zero */
            }
        fclose(fpses);
        printf("%d\n",n);
        for(i=0;i<n;i++) printf("%g,%g,%g\n",f[i],g[i],p[i]);
        fflush(stdout);
}
```

divide.c

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 500   /* Maximum line length */
#define MAXNAME 11    /* Maximum file name length */
#define MAXCHAN 8     /* Maximum Number of channels */

/* "divide" is used on files produced by the controller box if more than
   one channel is active. The file must be "separate"d first. The output
   files have names with an extension according to the number of the
   channel. If an option "-b" is specified, a special file naming
   convention is used: channel 1 starts with 'p', channel 3 starts with
   'f'. Source file in this case must start with 'b'.

   usage: divide [-b] <file>

        Yanko Sheiretov          12/8/92          Revised:  8/20/93          */

main(argc, argv)
        int argc;
        char *argv[];
{
        char line[MAXLINE], name[MAXNAME], flag=0, fline[MAXLINE];
        char fname[MAXNAME], ext[5];
        static char stat[MAXCHAN] = {0,0,0,0,0,0,0,0};
        FILE *fin, *fout[MAXCHAN];
        int i, ch;

        if (argc != 2 && argc != 3) {
          fprintf(stderr, "Usage:  \tdivide <file>\n\tdivide -b <bfile>\n");
                exit(); }
        if (argc == 3){
                if (strcmp(argv[1], "-b")!=0 || *argv[2] != 'b'){
          fprintf(stderr, "Usage:  \tdivide <file>\n\tdivide -b <bfile>\n");
                        exit(); }
```

```
        else flag = 1;}
strcpy(name, argv[1+flag]);

if ((fin = fopen(name, "r")) == NULL) {
        fprintf(stderr, "divide:  \tcan't open file %s\n", name);
        exit(); }                                                    40

fgets(fline, MAXLINE, fin);
while(1) {
        fgets(line, MAXLINE, fin);
        sscanf(line+4, "%d", &ch);
        if (stat[ch]==1) break;
        stat[ch]=1;
        if (flag) {
                if (ch!=1 && ch!=3)
         fprintf (stderr, "-b option allows only channels 1 & 3\n");  50
              else {
                        *fname=ch==1?'p':'f';
                        *(fname+1) = '\0';
                        strcat(fname,name+1); }}
        else {
                strcpy(fname,name);
                sprintf(ext, ".%d",ch);
                strcat(fname,ext);}
        if ((fout[ch]=fopen(fname, "w")) == NULL) {
            fprintf(stderr, "divide:  \tcan't open file %s\n", fname);  60
                exit(); }
        fputs(fline, fout[ch]);
        fputs(line, fout[ch]); }

do {
        if (sscanf(line+4,"%d",&ch) != 1) continue;
                        /* ignore bad lines */
        fputs(line, fout[ch]);}
while (fgets(line, MAXLINE, fin) != NULL);
                                                                     70
```

```
        for(i=0;i<MAXCHAN;i++) if (stat[i]==1) fclose(fout[i]);
        fclose (fin);
}
```

```
/* This is a useful little program which manipulates pairs of data.
   It takes its input from stdin and writes its output to stdout.
   The command line should indicate what to do with the numbers.
   Key letters are: a (add) followed by a number, add the number;
   s (scale) scale the number; l (log10) take its log10;
   p (pow10) take its antilog10; n (nothing) leave number as is.
   Some examples:
        do l a -1.0 < infile > outfile
        do n l < infile > outfile
        do a 3.14 s 2.75 < infile > outfile                          10
        extrapolate < infile | kk | do l n > outfile

                Yanko Sheiretov, 7/28/92        1/25/94        */

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

main(argc, argv)
        int argc;
        char *argv[];                                                20

{
        double x, y;
        char funa, funb, flag=0;
        double numa, numb;

        if (argc<3) goto usage;
        funa = *argv[1];
        if (funa!='a' && funa!='s' && funa!='l' && funa!='p' && funa!='n')
                goto usage;                                          30
        if (funa=='a' || funa=='s')
        {
                if (sscanf(argv[2], "%lf", &numa) != EOF) flag = 1;
                else goto usage;
```

```
        }
        if (flag && argc < 4) goto usage;
        funb = *argv[2+flag];
        if (funb!='a' && funb!='s' && funb!='l' && funb!='p' && funb!='n')
                goto usage;
        if (funb=='a' || funb=='s')                                          40
        {
                if(argc!=4+flag || sscanf(argv[3+flag], "%lf", &numb)==EOF)
                        goto usage;
        }
        else if (argc!=3+flag) goto usage;

        while (scanf("%lf %lf", &x, &y) != EOF)
                printf("%g\t%g\n",
                        funa=='a' ? x+numa :
                        funa=='s' ? x*numa :                                 50
                        funa=='l' ? log10(x) :
                        funa=='p' ? pow(10.0, x) : x,
                        funb=='a' ? y+numb :
                        funb=='s' ? y*numb :
                        funb=='l' ? log10(y) :
                        funb=='p' ? pow(10.0, y) : y);
        exit(0);
    usage:
        fprintf(stderr, "usage:  do fl [num] fr [num]\n");
        fprintf(stderr, "\tfl and fr can be:\n\t a (add), s (scale), l (log10), p (pow(10, x)), and n
        fprintf(stderr, "\tInput is from stdin and output is to stdout\n");

}
```

---

domerge.c

```
/* This program takes as an argument a template data file which consists of
   pairs of numbers on individual lines. It outputs to stdout in the same
   format, but with the first number replaced by a number read from stdin.

   Yanko Sheiretov,                6/9/93                                  */

#include <stdio.h>

main(argc, argv)
        int argc;                                                          10
        char *argv[];

{
        double x, y, z;
        FILE *fp;

        if (argc != 2) {
          fprintf(stderr, "usage:  \tdomerge tmplfile < infile > outfile\n");
          exit(); }
                                                                           20
        if ((fp = fopen(argv[1], "r")) == NULL) {
          fprintf(stderr, "domerge:  \tcannot open file %s\n", argv[1]);
          exit(); }
        while (scanf("%lf", &x)==1 && fscanf(fp, "%lf %lf", &y, &z)==2)
          printf("%g\t%g\n", x, z);
        fclose(fp);
}
```

nothing.c

```
#include <stdio.h>

/* This program takes input from stdin and does nothing with it
   Yanko Sheiretov                5/4/94                          */

main()
{
        while (getchar() != EOF);
}
```

10

only.c

```
#include <stdio.h>
#define MAXNUM 100

/* 'Only' cuts off parts of a data file. <lb> data points from the end
   are discarded. On the front side, all data points with x-coordinate
   greater than <rb> are also discarded. In this sense <lb> and <rb>
   ARE NOT equivalent - one is an integer and the other is a float.
   It is assumed that the x-coordinates are in DESCENDING order.

   Usage: only <file> <lb> <rb>                                          10

        Yanko Sheiretov          8/12/92          1/25/94                */

main(argc, argv)
        int argc;
        char *argv[];

{
        double x[MAXNUM], y[MAXNUM];
        int lb, i=0, j;                                                  20
        double rb;
        FILE *fp;

        if (argc != 4 ||
           sscanf(argv[2], "%d", &lb) == 0 ||
           sscanf(argv[3], "%lf", &rb) == 0) {
                fprintf(stderr, "Usage:  \tonly <file> <lb> <rb>\n");
                exit (); }
        if ((fp = fopen(argv[1], "r")) == NULL) {
                fprintf(stderr, "only:  \tcan't open file %s\n", argv[1]);  30
                exit(); }
        while (fscanf(fp, "%lf %lf",x+i,y+i) != EOF) i++;
        fclose(fp);
        if ((fp = fopen(argv[1], "w")) == NULL) {
```

```
            fprintf(stderr, "only:  \tcan't open file %s\n", argv[1]);
            exit(); }
        for (j=0;j<i-lb;j++)
            if(x[j]<=rb)
                fprintf(fp, "%g\t%g\n", x[j], y[j]);
        fclose(fp);
}
```

rev.c

```
/* This program reverses a file */

main ()
{
        double x[100], y[100];
        int i=0;

        while (scanf("%lf %lf", x+i, y+i) == 2) i++;
        i--;
        for(; i>=0; i--) printf("%g\t%g\n", x[i], y[i]);
}
```

separate.c

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 500   /* Maximum line length */
#define MAXNAME 11   /* Maximum file name length */

/* "separate" takes a file which contains output of the controller box and
   separates it into individual files which contain one set of measurements
   each. The original file must end on 'x' and the new files have the 'x'
   substituted with consecutive letters of the alphabet. In order for this
   to work, the original [gd] command must be on a new line. An optional
   argument specifies the maximum number of files to be written.

   usage: separate <file> [max]

                Yanko Sheiretov          10/8/92          6/17/93 */

main(argc, argv)
        int argc;
        char *argv[];
{
        char line[MAXLINE], name[MAXNAME], flag=1, *ind;
        FILE *fin, *fout;
        int max;

        if (argc != 2 && argc != 3) {
                fprintf(stderr, "Usage:   \tseparate <filex> [max]\n");
                exit(); }
        strcpy(name, argv[1]);

        ind = name;
        while(*ind++ != '\0');   /* Find the end of the string */
        ind -= 2;
        if (*ind != 'x') {          /* File must end on 'x' */
                fprintf(stderr, "separate:   \tfile must end on 'x'\n");
```

```
            exit(); }

        if ((fin = fopen(name, "r")) == NULL) {
                fprintf(stderr, "separate:  \tcan't open file %s\n", name);
                exit(); }
        if (argc == 2) max = 23;                                              40
        else {
                sscanf(argv[2], "%d", &max);
                if (max > 23) {
                  fprintf(stderr, "Number of files set to a maximum of 23\n");
                  max = 23; } }
        *line = '\0';
        for (*ind='a';flag && *ind-'a'<max;(*ind)++) {
                if ((fout = fopen(name, "w")) == NULL) {
                        fprintf(stderr, "separate:  \tcan't open file %s\n", name);
                        exit(); }                                             50
                fprintf(stderr, "Writing %s ...", name);
                if (*line != '\0') fputs(line, fout);
                while (1) {
                        if (fgets(line, MAXLINE, fin) == NULL) {
                                flag = 0;
                                break; }
                        if (*line !='[') continue;
                        if ((line[1]=='G' || line[1]=='g') &&
                           (line[2]=='D' || line[2]=='d'))
                                fputs(line, fout);                            60
                        if (line[1]=='G' && line[2]=='H') break; }
                fclose (fout);
                fprintf(stderr, "done\n"); }
        fclose (fin);
}
```

G.4 High-Level Data Processing

inv.c

```
# include       <stdio.h>
# include       <math.h>
# define        PI      3.14159265      /* PI */
# define        EOL     '\n'            /* end of line definition */

/* "inv" takes a file produced by the controller box and outputs
   two files containing data for epsilon' and epsilon" with
   extensions .e1 and .e2 respectively. An optional second argument
   specifies a file which contains information about the interface
   box. The default is /u/yanko/.invsetup                              10

        Yanko Sheiretov     Documented: 12/09/92   Revised: 1/25/94  */

main(argc, argv)
int     argc;
char    *argv[];

{
        char    fout1[11], fout2[11];
        char    junk;                                                   20
        FILE    *fpo1,*fpo2, *fpses, *fpset;
        double  r2, c2, cair, ch, r1, c1, f, g, p, o, dt, dg, dp;
        int n;

        if (argc != 2 && argc != 3) {
                fprintf(stderr, "Usage:  \tinv <file> [setfile]\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {                     30
            fprintf(stderr, "Error:  cannot open %s\n",argv[1]);
            exit(1);
```

```
    }

if ((fpset = fopen(argc==2?"/u/yanko/.invsetup":argv[2], "r"))==NULL) {
        fprintf(stderr, "Error:  cannot open %s\n", argc==2?"/u/yanko/.invsetup":argv[2]);
        exit(); }

strcpy(fout1, argv[1]);
strcat(fout1, ".e1");                                                    40
strcpy(fout2, argv[1]);
strcat(fout2, ".e2");

fscanf(fpset,"%lf",&r2);
while((junk = fgetc(fpset))!= EOL);
fscanf(fpset,"%lf",&c2);
while((junk = fgetc(fpset))!= EOL);
fscanf(fpset,"%lf",&cair);
fclose(fpset);
                                                                         50
fpo1 = fopen(fout1, "w");
fpo2 = fopen(fout2, "w");
while((junk=fgetc(fpses)) != EOL && junk != EOF); /* skip header */

n = 0;
while(1) {
        do junk=fgetc(fpses);
    while (junk != EOF && junk != EOL && junk != ']' && junk != ',');
        if (junk == EOF) break;
        if (junk == EOL || junk == ']') continue;                        60
        fscanf(fpses,"%lf,%lf,%lf,%lf,%lf",&ch,&f,&g,&p,&o);
        n++;
        dt = 2.*PI*pow(10.,f)*r2*c2*1.e-3;
        dg = pow(10., g/20.);
        dp = p*PI/180.;
        c1 = c2*dg*(cos(dp)-dg+sin(dp)/dt)/(1+dg*dg-2.*dg*cos(dp));
        r1 = r2*(1.+dg*dg-2.*dg*cos(dp))/(dg*(cos(dp)-dg-dt*sin(dp)));
        if (c1 > 0.) fprintf(fpo1, "%g\t%g\n", f, c1/cair);
```

```
            if (r1 > 0.) fprintf(fpo2, "%g\t%g\n", f, log10(1.e3/(r1*cair*2.*PI))-f);
            while(fgetc(fpses) != ']');                                          70
            }
        fclose(fpses);
        fclose(fpo1);
        fclose(fpo2);
        printf("Done ...  %d data points.\n",n);
        fflush(stdout);
}
```

.invsetup

This is the setup file required by **inv.c** and **rcinv.c**	
9.78	*Load resistance in GOhm*
123	*Load capacitance in pF*
14.6	*Air capacitance in pF (for* $d = 0.86$ *mm)*

lstsq.c

```
/* This function does a least-squares fit of a line to a set
   of data points. x & y are arrays of data, n is the number of
   points and slope and yint are pointers to locations where the
   results are to be written */

void lstsq(x,y,n,slope,yint)
        double *x, *y;
        int n;
        double *slope,*yint;
{                                                                    10
        double sx = 0.0, sy = 0.0, sxy = 0.0, sx2= 0.0, xx, yy;
        int i;
        for(i=0;i<n;i++)
        {
                xx=*(x+i);
                yy=*(y+i);
                sx+=xx;
                sy+=yy;
                sxy+=xx*yy;
                sx2+=xx*xx;                                          20
        }
        *slope = (n*sxy-sx*sy)/(n*sx2-sx*sx);
        *yint = sy/n - *slope*sx/n;
}
```

out2e.c

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 500   /* Maximum line length */
#define MAXNAME 11   /* Maximum file name length */

/* "out2e" takes as an input an output file of "parestsx" and produces
   two output data files containing the data for epsilon' and epsilon"
   with extensions .e1 and .e2 respectively, replacing the .out extension
   of the input file.

   usage: out2e <file.out>

                Yanko Sheiretov          12/9/92          R. 1/25/94        */

/* Revised so that commas between numbers are allowed in input file.
   Some Fortran compilers will cause these commas to appear in the
   output of parest.

                Yanko Sheiretov          5/7/94                             */

main(argc, argv)
        int argc;
        char *argv[];
{
        char line[MAXLINE];
        char name[MAXNAME];
        FILE *fin, *fout1, *fout2;
        double f, e1, e2;

        if (argc != 2) {
                fprintf(stderr, "Usage:  \tout2e <file.out>\n");
                exit(); }
        strcpy(name, argv[1]);
        if (strcmp(name+strlen(name)-4,".out")!=0){
```

```
            fprintf(stderr, "out2e:  \tInput file must end with .out\n");
            exit(); }

        if ((fin = fopen(name, "r")) == NULL) {
            fprintf(stderr, "out2e:  \tcan't open file %s\n", name);
            exit(); }                                                    40
        strcpy(name+strlen(name)-4, ".e1");
        if ((fout1 = fopen(name, "w")) == NULL) {
            fprintf(stderr, "out2e:  \tcan't open file %s\n", name);
            exit(); }
        strcpy(name+strlen(name)-1, "2");
        if ((fout2 = fopen(name, "w")) == NULL) {
            fprintf(stderr, "out2e:  \tcan't open file %s\n", name);
            exit(); }

        while (getc(fin) != '$');                                        50
        getc(fin);
        while ((fgets(line, MAXLINE, fin) != NULL) && *line != '$') {
            sscanf(line, "%lf%*c%lf%*c%lf", &f, &e1, &e2);
            fprintf(fout1, "%g\t%g\n", f, e1);
            fprintf(fout2, "%g\t%g\n", f, e2); }
        fclose (fout2);
        fclose (fout1);
        fclose (fin);
}
```

```
# include       <stdio.h>
# include       <math.h>
# define        PI      3.14159265      /* PI */
# define        EOL     '\n'            /* end of line definition */

/* "rcinv" takes a file produced by the controller box and outputs
   two files containing data for R and C with
   extensions .rr and .cc respectively. A file .invsetup with
   information on the interface box must exist in the home
   directory /u/yanko.                                              10
        Yanko Sheiretov         6/22/93         1/25/94         */

main(argc, argv)
int     argc;
char    *argv[];

{
        char    fout1[11],fout2[11];
        char    junk;
        FILE    *fpo1,*fpo2, *fpses, *fpset;                        20
        double  r2, c2, cair, ch, r1, c1, f, g, p, o, dt, dg, dp;
        int n;

        if (argc != 2) {
                fprintf(stderr, "Usage:   \trcinv <file>\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {
            fprintf(stderr, "Error:   cannot open %s\n",argv[1]);   30
            exit(1);
            }

        if ((fpset = fopen("/u/yanko/.invsetup", "r")) == NULL) {
```

```
        fprintf(stderr, "Error:  cannot open .invsetup\n");
        exit(); }

strcpy(fout1, argv[1]);
strcat(fout1, ".cc");
strcpy(fout2, argv[1]);                                          40
strcat(fout2, ".rr");

fscanf(fpset,"%lf",&r2);
while((junk = fgetc(fpset))!= EOL);
fscanf(fpset,"%lf",&c2);
while((junk = fgetc(fpset))!= EOL);
fscanf(fpset,"%lf",&cair);
fclose(fpset);

fpo1 = fopen(fout1, "w");                                        50
fpo2 = fopen(fout2, "w");
while((junk=fgetc(fpses))!=EOL);
for(n=0;(junk=fgetc(fpses))!=EOF;++n) {
        junk=fgetc(fpses);
        while((junk!=',') && (junk!=']')) {
                junk=fgetc(fpses);
                }
        if (junk==',') {
        fscanf(fpses,"%lf,%lf,%lf,%lf,%lf",&ch,&f,&g,&p,&o);
        dt = 2.*PI*pow(10.,f)*r2*c2*1.e-3;                       60
        dg = pow(10., g/20.);
        dp = p*PI/180.;
        c1 = c2*dg*(cos(dp)-dg+sin(dp)/dt)/(1+dg*dg-2.*dg*cos(dp));
        r1 = r2*(1.+dg*dg-2.*dg*cos(dp))/(dg*(cos(dp)-dg-dt*sin(dp)));
        fprintf(fpo1, "%g\t%g\n", f, c1);
        fprintf(fpo2, "%g\t%g\n", f, r1); }
        while((junk=fgetc(fpses))!=EOL);
        }
fclose(fpses);
fclose(fpo1);                                                    70
```

```
        fclose(fpo2);
        printf("Done ...  %d data points.\n",n-1);
        fflush(stdout);
}
```

G.5 Data Interpretation

extrapolate.c

```
/*      Yanko Sheiretov          7/28/92                    */

#include <stdio.h>
#include <math.h>
#define MAX 100

#include "Lib/lstsq.c"
#include "Lib/reverse.c"
#include "Lib/maximum.c"

/* This program takes a set of data representing the dependence of
   epsilon" on frequency and extrapolates the data assuming values
   the slope on either side of the peak to be of equal magnitude and
   opposite sign. The data should then be easy to integrate using
   Kramers-Kronig relations to obtain values for epsilon'. The data
   is read from stdin and output to stdout. The value of the slope
   is printed out to stderr.                                    */

main()
{
        double fin[MAX], ein[MAX];      /* both in log10 form */
        int i=0, maxi, shift, j;
        double slope, yint, peak, f;

        while(scanf("%lf %lf", fin+i, ein+i) != EOF) i++;
        reverse(fin,i);
        reverse(ein,i);

        maxi = maximum(ein, i);

        lstsq(fin+maxi+5, ein+maxi+5, i>maxi+25?20:i-maxi-5, &slope, &yint);
        peak = yint+slope*fin[maxi];
```

```
        fprintf(stderr, "m = %f\n", -slope);

        shift=30-maxi;

        for(j=0,f=fin[maxi]-0.1*shift;j<shift-5;j++,f+=0.1)
          printf("%f\t%f\n",pow(10.0,f),pow(10.0,peak-slope*(f-fin[maxi])));
        for(j=0;j<(i>maxi+30?maxi+30:i);j++)                                    40
                printf("%f\t%f\n",pow(10.0,fin[j]),pow(10.0,ein[j]));
        if (j<31+maxi)
                for(f=fin[j-1]+0.1;j<31+maxi;j++,f+=0.1)
                        printf("%f\t%f\n",pow(10.0,f),pow(10.0,yint+slope*f));
}
```

```
/* This program fits a power-law curve to a set of data supplied
   from stdin. It has an optional argument specifying the value of
   m. The program outputs the values of k and fp to stderr and pairs
   of data points along the fitted curve to stdout. The function
   of the curve being fitted is:

   y = k/(pow(x/fp,m)+pow(fp/x,m))     i.e.

                k
   y = ------------------------                                    10
         x  m     fp m
       (---)  + (---)
        fp       x

                        Yanko Sheiretov          7/29/92
                        Last updated             8/13/92          */
#include <stdio.h>
#include <math.h>
#define MAX 100
                                                                   20
#include "Lib/power_fit.c"

main(argc,argv)
        int argc;
        char *argv[];
{
        double fin[MAX], ein[MAX];        /* both in log10 form */
        int i=0;
        double f, e, k, fp, m=0.5, j, ee;
                                                                   30
        if(argc>1) sscanf(argv[1], "%lf", &m);

        while(scanf("%lf %lf", &f, &e) != EOF)
         {
```

```
        fin[i]=pow(10.0,f);
        ein[i]=pow(10.0,e);
        i++;
      }

    power_fit(fin,ein,i,m,&k,&fp);                                    40

    fprintf(stderr,"K = %f\tfp = %f\n",k,fp);
    ee = ((int)(10.0*log10(fp)))/10.0 - 3.0;
    e = ee >= -5.0 ? ee : -5.0;
    f = pow(10.0, e);
    for(j=e;j<=e+6.0;j+=0.1,f=pow(10.0,j))
      printf("%f\t%f\n", f, k/(pow(f/fp,m)+pow(fp/f,m)));
}
```

---

fith.c

```
/* This program takes a reference .r2 file and a test .r2 file and
   computes by how much the latter would have to be shifted in frequency
   to produce a least squares sum. The input files will have to have been
   reversed with the program rev.

   Yanko Sheiretov          10/15/93        R. 11/18/93          */

#include <stdio.h>

main(argc, argv)                                                    10
        int argc;
        char *argv[];
{
        FILE    *f1, *f2;
        int     i, n, max, maxref, shift, step, flag=0;
        double  sum = 1.e6, oldsum = 1.e6, f, ff, ref[65], test[65];



        if (argc != 3) {
                fprintf(stderr, "usage:  \tfith <fref> <fin>\n");     20
                exit(); }
        if ((f1 = fopen(argv[1], "r")) == NULL) {
                fprintf(stderr, "Error:  \tCan't open %s \n", argv[1]);
                exit(); }
        if ((f2 = fopen(argv[2], "r")) == NULL) {
                fprintf(stderr, "Error:  \tCan't open %s \n", argv[2]);
                exit(); }
        for (maxref=0,f=-2.3; maxref < 64; maxref++, f+=0.1) {
                if (fscanf(f1, "%lf %lf", &ff, ref+maxref) !=2) break;
                if ((ff-f)*(ff-f) >= 0.0004) {                      30
                        fprintf(stderr, "Problems with input file %s \n", argv[1]);
                        exit(); } }
        for (max=0,f=-2.3; max < 64; max++, f+=0.1) {
                if (fscanf(f2, "%lf %lf", &ff, test+max) !=2) break;
```

```
        if ((ff-f)*(ff-f) >= 0.0004) {
                fprintf(stderr, "Problems with input file %s \n", argv[2]);
                exit(); } }


    for(shift=0,step=1,flag=0; 1; shift += step) {                          40
        oldsum = sum;
        for(sum = 0.0, n=0, i=0; i < maxref && i+shift < max; i++) {
            if (i+shift >= 0) {
                sum += (ref[i]-test[i+shift])*(ref[i]-test[i+shift]);
                n++; } }
        sum /= (double)n;
        if (sum > oldsum) {
            if (flag) break;
            else {
                step *= -1;                                                 50
                flag = 1; }}}
    shift -= step;
    printf("frequency shift is %f\n", (double)shift/10.0);
}
```

fitm.c

```
/* Aversion of 'fit' which also attempts to fit a value for m.
   This program fits a power-law curve to a set of data supplied
   from stdin. The program outputs the values of K, fp, and m to
   stderr and pairs of data points along the fitted curve to
   stdout. The function of the curve being fitted is:

   y = k/(pow(x/fp,m)+pow(fp/x,m))     i.e.

            k
   y = ------------------
      x  m    fp m
     (---)  + (---)
      fp       x

                        Yanko Sheiretov          7/29/92
                        Last updated             8/13/92          */

#include <stdio.h>
#include <math.h>
#define MAX 100

#include "Lib/power_fit.c"

main()
{
        double fin[MAX], ein[MAX];      /* both in log10 form */
        int i=0;
        double f, e, k, fp, m=0.5, j, oldlsq, newlsq, ee;

        while(scanf("%lf %lf", &f, &e) != EOF)
          {
            fin[i]=pow(10.0,f);
            ein[i]=pow(10.0,e);
            i++;
```

```
  }

newlsq = power_fit(fin,ein,i,m,&k,&fp);
do {
        m -= 0.005;
        oldlsq = newlsq;                                        40
        newlsq=power_fit(fin,ein,i,m,&k,&fp); }
while (newlsq < oldlsq);
m += 0.005;
if (m == 0.5) {
        newlsq = oldlsq;
        do {
                m += 0.005;
                oldlsq = newlsq;
                newlsq=power_fit(fin,ein,i,m,&k,&fp); }
        while (newlsq < oldlsq);                                50
        m -= 0.005; }

fprintf(stderr,"K = %f\tfp = %f\tm = %f\n",k,fp,m);
ee = ((int)(10.0*log10(fp)))/10.0 - 3.0;
e = ee >= -5.0 ? ee : -5.0;
f = pow(10.0, e);
for(j=e;j<=e+6.0;j+=0.1,f=pow(10.0,j))
  printf("%f\t%f\n", f, k/(pow(f/fp,m)+pow(fp/f,m)));
}
                                                                60
```

```
/* This program uses the Kramers–Kronig Relations to calculate e" from
   e'. Data is read from stdin and output to stdout. The integration
   contains a singularity about the point of frequency being calculated.
   Therefore the integration is done in three parts: a small interval df
   on either side of f is neglected, because the area around the singularity
   cancels out. For this cancellation to be valid, the integrand must stay
   constant. Therefore df is taken to be one step of frequency to the left.
   The integration is then carried out independently to the left and to the
   right of this interval. Since df would be greater on the right side of the
   singularity, due to the logarithmic step in frequency, there is a need
   to add on the thin sliver equal to the difference between the two df's on
   either side. The numerical calculation uses the trapezoidal rule.

   Yanko Sheiretov          12/16/92        12/16/92          */

#include <stdio.h>
#define MAX 100
#define PI 3.14159265

main()
{
        double fin[MAX], ein[MAX];
        int n=0, i, j;
        double f, x, xh, xl, sl, sr, sm;

        while(scanf("%lf %lf", fin+n, ein+n) != EOF) n++;

        for(i=2;i<n-2;i++)        /* Main cycle in f */
        {
           f = *(fin+i);
           sl = 0.0;
           sm = 0.0;
           sr = 0.0;
           for(j=i-1;j>0;j--)     /* Left side of integral */
```

```
            {
                    xh = fin[j];
                    xl = fin[j-1];
                    sl += (ein[j]/(xh*xh-f*f) +
                            ein[j-1]/(xl*xl-f*f))*(xh-xl);
            }                                                    40

            /* Now find the position of the singularity sliver */
            xh = fin[i+1];
            xl = 2.0*f - fin[i-1];
            sm = (ein[i+1]/(xh*xh-f*f) +
                  (((xl-f)*ein[i+1]+(xh-xl)*ein[i])/(xh-f))/
                  (xl*xl-f*f))*(xh-xl);

            for(j=i+1;j<n-1;j++)          /* Right side of integral */
            {                                                    50
                    xh = fin[j+1];
                    xl = fin[j];
                    sr += (ein[j+1]/(xh*xh-f*f) +
                            ein[j]/(xl*xl-f*f))*(xh-xl);

            }
            printf("%f\t%f\n", f, -f*(sl+sm+sr)/PI);
        }
}
```

kk21.c

```
/* This program uses the Kramers–Kronig Relations to calculate e' from
   e". Data is read from stdin and output to stdout. The integration
   contains a singularity about the point of frequency being calculated.
   Therefore the integration is done in three parts: a small interval df
   on either side of f is neglected, because the area around the singularity
   cancels out. For this cancellation to be valid, the integrand must stay
   constant. Therefore df is taken to be one step of frequency to the left.
   The integration is then carried out independently to the left and to the
   right of this interval. Since df would be greater on the right side of the
   singularity, due to the logarithmic step in frequency, there is a need
   to add on the thin sliver equal to the difference between the two df's on
   either side. The numerical calculation uses the trapezoidal rule.     */

#include <stdio.h>
#define MAX 100
#define PI 3.14159265

main()
{
        double fin[MAX], ein[MAX];
        int n=0, i, j;
        double f, x, xh, xl, sl, sr, sm;

        while(scanf("%lf %lf", fin+n, ein+n) != EOF) n++;

        for(i=2;i<n-2;i++)        /* Main cycle in f */
        {
           f = *(fin+i);
           sl = 0.0;
           sm = 0.0;
           sr = 0.0;
           for(j=i-1;j>0;j--)     /* Left side of integral */
                {
                        xh = fin[j];
```

```
                xl = fin[j-1];
                sl += (xh*ein[j]/(xh*xh-f*f) +
                        xl*ein[j-1]/(xl*xl-f*f))*(xh-xl);
        }

        /* Now find the position of the singularity sliver */
        xh = fin[i+1];
        xl = 2.0*f - fin[i-1];
        sm = (xh*ein[i+1]/(xh*xh-f*f) +
              xl*(((xl-f)*ein[i+1]+(xh-xl)*ein[i])/(xh-f))/
              (xl*xl-f*f))*(xh-xl);

        for(j=i+1;j<n-1;j++)          /* Right side of integral */
        {
                xh = fin[j+1];
                xl = fin[j];
                sr += (xh*ein[j+1]/(xh*xh-f*f) +
                        xl*ein[j]/(xl*xl-f*f))*(xh-xl);

        }
        printf("%f\t%f\n", f, (sl+sm+sr)/PI);
    }
}
```

maximum.c

```
/* This function takes an array of doubles (x) and the size of the
   array (n) and returns the index number for the largest number in
   the array. */

int maximum(x,n)
    double *x;
    int n;
{
 double max;
 int i, r=0;                                                    10

 max = *x;
 for(i=1;i<n;i++)
  if (*(x+i)>max)
    {
        max = *(x+i);
        r = i;
    }
 return r;
}                                                               20
```

```
#include <math.h>

/* This function performs a non-linear least-squares-fit to a data set
   according to the following formula:

            k
   y = ------------------
         x  m     fp m
       (---)  + (---)
         fp       x                                                    10

   The results are written into the locations k and fp. The function
   returns the sum of the squared errors

                   Yanko Sheiretov          8/6/92                */

double power_fit(x,y,n,m,k,fp)
        double *x, *y;
        int n;
        double m, *k, *fp;                                            20
{
        double spq=0.0, spoq=0.0, sq2=0.0, soq2=0.0, sp2=0.0;
        double p, q, r, s;
        int i;
        for(i=0;i<n;i++)
        {
                q=pow(*(x+i), m);
                p=1.0/(*(y+i));
                sp2 += p*p;
                spq += p*q;                                           30
                spoq += p/q;
                sq2 += q*q;
                soq2 += 1.0/(q*q);
        }
```

```
        s = sqrt((spq*soq2-n*spoq)*(spoq*sq2-n*spq))/(sq2*soq2-n*n);
        r = sqrt(((spoq*sq2-n*spq)/(spq*soq2-n*spoq)));
        *k = 1.0/s;
        *fp = pow(r, 1.0/m);
        return (sp2 + s*s/(r*r)*sq2 + s*s*r*r*soq2 + 2.0*s*s*n -
                2*s*spq/r - 2*s*r*spoq);                                    40
}
```

reverse.c

```
/* This function reverses the order of elenemts in an array of doubles.
   x is a pointer to the array and n is the number of elements. */

void reverse(x,n)
    double *x;
    int n;
{
 double temp;
 int i;

                                                                    10
 for(i=0;i<n/2.0;i++)
  {
   temp = *(x+i);
   *(x+i) = *(x+n-i-1);
   *(x+n-i-1) = temp;
  }
}
```

sen12.c

```
/* sen12 is a program which plots the relative sensitivity of the
   inversion process (from z1 to z2) to variations in the phase
   data. Its output is the file "out", which is ready to be "plop"ped.

                Yanko Sheiretov                 9/18/92                 */

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

#define MAXPTS 500                                                      10
#define PI      3.14159265
#define UPPER 1.3
#define LOWER -1.3
#define CENTER -1.
#define LOWER2 -3.
#define SMALL 1.e-6
#define DO(x,ref) ((x)>UPPER+(ref)?UPPER+(ref):(x)<LOWER+(ref)?LOWER+(ref):(x))
#define DO2(x) ((x)>0?0:(x)<LOWER2?LOWER2:(x))
#define RND(x) ((x)-floor(x)<0.5?floor(x):ceil(x))
                                                                        20
main()
{
        double f[MAXPTS], m[MAXPTS], p[MAXPTS], w[MAXPTS];
        double x, refa, refb;
        FILE    *fout;
        int     n, i;
        double c1, c2, g1, g2, fl, fh, ww;

        fout = fopen("out", "w");
                                                                        30
        printf("Enter the value for C1 [pF]: \t\t");
        scanf("%lf", &c1);
        printf("Enter the value for C2 [pF]: \t\t");
        scanf("%lf", &c2);
```

```
printf("Enter the value for R1 [GOhm]:  \t\t");
scanf("%lf", &g1);
printf("Enter the value for R2 [GOhm]:  \t\t");
scanf("%lf", &g2);
printf("Enter the low frequency end [log]:  \t");
scanf("%lf", &fl);                                                    40
printf("Enter the high frequency end [log]:  \t");
scanf("%lf", &fh);

fprintf(fout, "window vertical 1/3\n");
fprintf(fout,"plot cross green marker 0.08\n");
fprintf(fout,"title top \"Inversion Process\"\n");
fprintf(fout, "label top \"R_1=%8.2fG\\Omega  R_2=%8.2fG\\Omega  C_1=%8.2fpF  C_2=%8.2fpF\"\n
        g1, g2, c1, c2);

c1 *= 1.e-12;                                                         50
c2 *= 1.e-12;
g1 = 1.e-9 / g1;
g2 = 1.e-9 / g2;
fl = floor(10.*fl)/10.;
fh = ceil(10.*fh)/10.;

for(n = 0; fl <= fh; fl += 0.1, n++) {
        f[n] = fl;
        w[n] = ww = 2.*PI*pow(10.,fl);
        m[n] = sqrt((g1*g1+c1*c1*ww*ww) /                             60
                        ((g1+g2)*(g1+g2)+(c1+c2)*(c1+c2)*ww*ww));
        p[n] = atan(c1*ww/g1) - atan((c1+c2)*ww/(g1+g2)); }

fprintf(fout, "label left \"\\cross Gain [dB]\"\n");
fprintf(fout, "label right \"\\circle Phase [deg]\"\n");
fprintf(fout, "plot cross green marker 0.08\n");
for(i=0;i<n;i++) fprintf(fout, "%f\t%f\n", f[i], 20.*log10(m[i]));

fprintf(fout, "plot circles green marker 0.08 use right\n");
for(i=0;i<n;i++) fprintf(fout, "%f\t%f\n", f[i], p[i]*180./PI);       70
```

```
fprintf(fout, "window vertical 1/3\n");
fprintf(fout, "label top \"Estimate with M and P rounded to 2 significant digits\"\n");
fprintf(fout, "label left \"\\cross log(\\alpha = R_1/R_2)\"\n");
fprintf(fout, "label right \"\\circle log(\\beta = C_2/C_1)\"\n");
fprintf(fout, "plot cross green marker 0.08\n");
refa = log10(g2/g1);
refb = log10(c2/c1);
for(i=0;i<n;i++) {
        p[i] = RND(100.*p[i])/100.;                                          80
        m[i] = RND(100.*m[i])/100.;
        x = (cos(p[i])+c1/g1*w[i]*sin(p[i]))/m[i] - 1.;
        x = x<=0.?refa-log10(SMALL):log10(x);
        fprintf(fout, "%f\t%f\n", f[i], DO(x,refa)); }
fprintf(fout, "plot circles green marker 0.08\n");
for(i=0;i<n;i++) {
        x = (cos(p[i])-g1/c1/w[i]*sin(p[i]))/m[i] - 1.;
        x = x<=0.?refb-log10(SMALL):log10(x);
        fprintf(fout, "%f\t%f\n", f[i], DO(x,refb)); }
fprintf(fout, "plot line green\n");                                          90
fprintf(fout, "%f\t%f\n%f\t%f\n", f[0], refa, f[n-1], refa);
fprintf(fout, "plot line green\n");
fprintf(fout, "%f\t%f\n%f\t%f\n", f[0], refb, f[n-1], refb);


fprintf(fout,"window vertical 1/3\n");
fprintf(fout,"label top \"Relative Phase Sensitivity [log(deg^{-1})]\"\n");
fprintf(fout,"label left \"\\cross \\alpha; \\circle \\beta\"\n");
fprintf(fout,"label bottom \"log(freq)\"\nlabel right \"Percent\"\n");
fprintf(fout, "y axis scale %f %f\n", LOWER2, 0.);                          100
fprintf(fout, "z axis log scale %f %f\n", pow(10.,2+LOWER2), 100.);
fprintf(fout,"plot cross green marker 0.08\n");
for(i=0;i<n;i++) {
        x = fabs(((c1+c2)/g1*w[i]-1.)*PI/180.*g1/g2);
        x = log10(x==0.?SMALL:x);
        fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }
```

```
        fprintf(fout,"plot circles green marker 0.08\n");
        for(i=0;i<n;i++) {
                x = fabs(((g1+g2)/c1/w[i]+1.)*PI/180.*c1/c2);
                x = log10(x==0.?SMALL:x);                                    110
                fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }
        fprintf(fout,"plot line green\n");
        fprintf(fout,"%f\t%f\n%f\t%f\n",f[0], CENTER, f[n-1], CENTER);
        printf("Done.\n");
        fclose(fout);
}
```

sen21.c

```
/* sen21 is a program which plots the relative sensitivity of the
   inversion process (from z2 to z1) to variations in the phase and
   magnitude data. Its output is the file "out", which is ready to be
   "plop"ped.

                Yanko Sheiretov                 9/22/92                 */

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

#define MAXPTS 500
#define PI      3.14159265
#define UPPER 1.3
#define LOWER -1.3
#define CENTER -1.
#define LOWER2 -3.
#define SMALL 1.e-6
#define DO(x,ref) ((x)>UPPER+(ref)?UPPER+(ref):(x)<LOWER+(ref)?LOWER+(ref):(x))
#define DO2(x)  ((x)>0?0:(x)<LOWER2?LOWER2:(x))
#define RND(x)  ((x)-floor(x)<0.5?floor(x):ceil(x))

main()
{
        double  f[MAXPTS], m[MAXPTS], p[MAXPTS], w[MAXPTS];
        double  x, refa, refb;
        FILE    *fout;
        int     n, i;
        double  c1, c2, g1, g2, fl, fh, ww, mm, pp;
        static  long    idum = 700302;
        float   ran2();

        fout = fopen("out", "w");

        printf("Enter the value for C1 [pF]: \t\t");
```

```
scanf("%lf", &c1);
printf("Enter the value for C2 [pF]: \t\t");
scanf("%lf", &c2);
printf("Enter the value for R1 [GOhm]:  \t\t");
scanf("%lf", &g1);
printf("Enter the value for R2 [GOhm]:  \t\t");                    40
scanf("%lf", &g2);
printf("Enter the low frequency end [log]:  \t");
scanf("%lf", &fl);
printf("Enter the high frequency end [log]:  \t");
scanf("%lf", &fh);

fprintf(fout, "window vertical 1/4\n");
fprintf(fout,"plot cross green marker 0.08\n");
fprintf(fout,"title top \"Reverse Inversion Process\"\n");
fprintf(fout, "label top \"R_1=%8.2fG\\Omega  R_2=%8.2fG\\Omega  C_1=%8.2fpF 50C_2=%8.2fpF\"\n
        g1, g2, c1, c2);

c1 *= 1.e-12;
c2 *= 1.e-12;
g1 = 1.e-9 / g1;
g2 = 1.e-9 / g2;
fl = floor(10.*fl)/10.;
fh = ceil(10.*fh)/10.;

for(n = 0; fl <= fh; fl += 0.1, n++) {                             60
        f[n] = fl;
        w[n] = ww = 2.*PI*pow(10.,fl);
        m[n] = sqrt((g1*g1+c1*c1*ww*ww) /
                        ((g1+g2)*(g1+g2)+(c1+c2)*(c1+c2)*ww*ww));
        p[n] = atan(c1*ww/g1) - atan((c1+c2)*ww/(g1+g2)); }

fprintf(fout, "label left \"\\cross Gain [dB]\"\n");
fprintf(fout, "label right \"\\circle Phase [deg]\"\n");
fprintf(fout, "plot cross green marker 0.08\n");
for(i=0;i<n;i++) fprintf(fout, "%f\t%f\n", f[i], 20.*log10(m[i]));  70
```

```
fprintf(fout, "plot circles green marker 0.08 use right\n");
for(i=0;i<n;i++) fprintf(fout, "%f\t%f\n", f[i], p[i]*180./PI);

fprintf(fout, "window vertical 1/4\n");
fprintf(fout, "label top \"Estimate with 5%% noise in M and P\"\n");
fprintf(fout, "label left \"\\cross log(\\alpha = R_2/R_1)\"\n");
fprintf(fout, "label right \"\\circle log(\\beta = C_1/C_2)\"\n");
fprintf(fout, "plot cross green marker 0.08\n");
refa = log10(g1/g2);                                                    80
refb = log10(c1/c2);
for(i=0;i<n;i++) {
        pp = ((ran2(&idum)-0.5)*0.2+1.)*p[i];
        mm = ((ran2(&idum)-0.5)*0.2+1.)*m[i];
        x = (cos(pp)-mm-c2/g2*w[i]*sin(pp))/(1./mm+mm-2*cos(pp));
        x = x<=0.?refa-log10(SMALL):log10(x);
        fprintf(fout, "%f\t%f\n", f[i], DO(x,refa)); }
fprintf(fout, "plot circles green marker 0.08\n");
for(i=0;i<n;i++) {
        pp = ((ran2(&idum)-0.5)*0.2+1.)*p[i];                           90
        mm = ((ran2(&idum)-0.5)*0.2+1.)*m[i];
        x = (cos(pp)-mm+g2/c2/w[i]*sin(pp))/(1./mm+mm-2*cos(pp));
        x = x<=0.?refb-log10(SMALL):log10(x);
        fprintf(fout, "%f\t%f\n", f[i], DO(x,refb)); }
fprintf(fout, "plot line green\n");
fprintf(fout, "%f\t%f\n%f\t%f\n", f[0], refa, f[n-1], refa);
fprintf(fout, "plot line green\n");
fprintf(fout, "%f\t%f\n%f\t%f\n", f[0], refb, f[n-1], refb);

fprintf(fout,"window vertical 1/4\n");                                  100
fprintf(fout,"label top \"Relative Magnitude Sensitivity [log(dB^{-1})]\"\n");
fprintf(fout,"label left \"\\cross \\alpha; \\circle \\beta\"\n");
fprintf(fout,"label right \"Percent\"\n");
fprintf(fout, "y axis scale %f %f\n", LOWER2, 0.);
fprintf(fout, "z axis log scale %f %f\n", pow(10.,2+LOWER2), 100.);
fprintf(fout,"plot cross green marker 0.08\n");
```

```
for(i=0;i<n;i++) {
        mm = m[i];
        pp = p[i];
        x = fabs(((mm*mm+1.)*cos(pp)-2.*mm-(1.-mm*mm)                    110
                *c2/g2*w[i]*sin(pp))/(1.+mm*mm-2.*mm*cos(pp))
                /(1.+mm*mm-2.*mm*cos(pp))*mm*log(10.)/20.*g2/g1);
        x = log10(x==0.?SMALL:x);
        fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }
fprintf(fout,"plot circles green marker 0.08\n");
for(i=0;i<n;i++) {
        mm = m[i];
        pp = p[i];
        x = fabs(((mm*mm+1.)*cos(pp)-2.*mm+(1.-mm*mm)
                *g2/c2/w[i]*sin(pp))/(1.+mm*mm-2.*mm*cos(pp))            120
                /(1.+mm*mm-2.*mm*cos(pp))*mm*log(10.)/20.*c2/c1);
        x = log10(x==0.?SMALL:x);
        fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }
fprintf(fout,"plot line green\n");
fprintf(fout,"%f\t%f\n%f\t%f\n",f[0], CENTER, f[n-1], CENTER);

fprintf(fout,"window vertical 1/4\n");
fprintf(fout,"label top \"Relative Phase Sensitivity [log(deg^{-1})]\"\n");
fprintf(fout,"label left \"\\cross \\alpha; \\circle \\beta\"\n");
fprintf(fout,"label bottom \"log(freq)\"\nlabel right \"Percent\"\n");    130
fprintf(fout, "y axis scale %f %f\n", LOWER2, 0.);
fprintf(fout, "z axis log scale %f %f\n", pow(10.,2+LOWER2), 100.);
fprintf(fout,"plot cross green marker 0.08\n");
for(i=0;i<n;i++) {
        mm = m[i];
        pp = p[i];
        x=fabs(((mm*mm*mm-mm)*sin(pp)+(-(mm*mm*mm+mm)*cos(pp)+2.*mm*mm)
         *c2/g2*w[i])/(1.+mm*mm-2.*mm*cos(pp))/(1.+mm*mm-2.*mm*cos(pp))
         *PI/180.*g2/g1);
        x = log10(x==0.?SMALL:x);                                        140
        fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }
fprintf(fout,"plot circles green marker 0.08\n");
```

```
        for(i=0;i<n;i++) {
                mm = m[i];
                pp = p[i];
                x=fabs(((mm*mm*mm-mm)*sin(pp)+((mm*mm*mm+mm)*cos(pp)-2.*mm*mm)
                 *g2/c2/w[i])/(1.+mm*mm-2.*mm*cos(pp))/(1.+mm*mm-2.*mm*cos(pp))
                 *PI/180.*c2/c1);
                x = log10(x==0.?SMALL:x);
                fprintf(fout,"%f\t%f\n", f[i], DO2(x)); }                    150
        fprintf(fout,"plot line green\n");
        fprintf(fout,"%f\t%f\n%f\t%f\n",f[0], CENTER, f[n-1], CENTER);
        printf("Done.\n");
        fclose(fout);
}

#define M 714025
#define IA 1366
#define IC 150889
                                                                             160
float ran2(idum)
long *idum;
{
        static long iy,ir[98];
        static int iff=0;
        int j;

        if (*idum < 0 || iff == 0) {
                iff=1;
                if ((*idum=(IC-(*idum)) % M) < 0) *idum = -(*idum);          170
                for (j=1;j<=97;j++) {
                        *idum=(IA*(*idum)+IC) % M;
                        ir[j]=(*idum);
                }
                *idum=(IA*(*idum)+IC) % M;
                iy=(*idum);
        }
        j=1 + 97.0*iy/M;
```

```
        if (j > 97 || j < 1) fprintf(stdout,"RAN2:  This cannot happen.");
        iy=ir[j];                                                          180
        *idum=(IA*(*idum)+IC) % M;
        ir[j]=(*idum);
        return (float) iy/M;
}

#undef M
#undef IA
#undef IC
```

G.6 Plotting

ecomp.c

```
/* This program when piped into plop plots the .e1 and .e2 files */
/* It compares two sets of .e files by plotting them together.    */
/* Revised 6/22/93        Yanko Sheiretov                         */

#include <stdio.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 3) {                                                    10
                fprintf(stderr, "Usage:  \tcomp <file1> <file2> | plop\n");
                exit();}
        printf("window top\n");
        printf("title top \"Calculated Complex \\epsilon:  Comparison\"\n");
        printf("label top \"Files:  %s and %s\"\n", argv[1], argv[2]);
        printf("label left \"\\epsilon':  \\cross %s  \\circle %s\"\n", argv[1], argv[2]);
        printf("plot cross green marker 0.08\n");
        printf("input \"%s.e1\"\n", argv[1]);
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.e1\"\n", argv[2]);                               20
        printf("window bottom\n");
        printf("label left \"log(\\epsilon''):  \\cross %s  \\circle %s\"\n", argv[1], argv[2]);
        printf("label bottom \"log(freq)\"\n");
        printf("plot cross green marker 0.08\n");
        printf("input \"%s.e2\"\n", argv[1]);
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.e2\"\n", argv[2]);
}
```

eplot.c

```
/* This program when piped into plop plots the .e1 and .e2 files
   Yanko Sheiretov          ?????    10/7/93                          */

#include <stdio.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2) {
                fprintf(stderr, "Usage:  \teplot <file> | plop\n");
                exit();}
        printf("window top\n");
        printf("title top \"Calculated Complex \\epsilon\"\n");
        printf("label top \"File:  %s\"\n", argv[1]);
        printf("label left \"\\epsilon'/\\epsilon_0\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("plot line cross green marker 0.08\n");
        printf("input \"%s.e1\"\n", argv[1]);

        printf("window bottom\n");
        printf("label left \"log(\\epsilon'')/\\epsilon_0)\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("plot line cross green marker 0.08\n");
        printf("input \"%s.e2\"\n", argv[1]);
}
```

eplot3.c

```
/* This program when piped into plop plots the .e1 and .e2 files
   of the three wavelengths of the 3-l sensor. The files
   name.1.e1      name.2.e1      name.3.e1
   name.1.e2      name.2.e2      name.3.e2
   must exist in that directory.

   Yanko Sheiretov        10/1/93                                  */

#include <stdio.h>
main(argc, argv)                                                    10
        int argc;
        char *argv[];
{
        if (argc != 2) {
                fprintf(stderr, "Usage:  \teplot3 <name>\n");
                exit();}
        printf("window top\n");
        printf("title top \"Complex \\epsilon with 3-\\lambda.  File:  %s\"\n",
                                                        argv[1]);
        printf("label top \"\\triangle:  \\lambda = 5 mm   \\square:");      20
        printf(" \\lambda = 2.5 mm   \\circle:  \\lambda = 1 mm\"\n");
        printf("label left \"\\epsilon'/\\epsilon_0\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("plot triangles green marker 0.08\n");
        printf("input \"%s.1.e1\"\n", argv[1]);
        printf("plot squares green marker 0.08\n");
        printf("input \"%s.2.e1\"\n", argv[1]);
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.3.e1\"\n", argv[1]);

                                                                    30
        printf("window bottom\n");
        printf("label left \"log(\\epsilon''/\\epsilon_0)\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("plot triangles green marker 0.08\n");
```

```
        printf("input \"%s.1.e2\"\n", argv[1]);
        printf("plot squares green marker 0.08\n");
        printf("input \"%s.2.e2\"\n", argv[1]);
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.3.e2\"\n", argv[1]);
}                                                                    40
```

eplotx.c

```
/* This program when piped into plop plots the .e1 and .e2 files.
   It differs from eplot in that it adds axis scale information.
   Yanko Sheiretov          10/7/93                                    */

#include <stdio.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2) {                                                    10
                fprintf(stderr, "Usage:  \teplotx <file> | plop\n");
                exit();}
        printf("window top\n");
        printf("title top \"Calculated Complex \\epsilon\"\n");
        printf("label top \"File:  %s\"\n", argv[1]);
        printf("label left \"\\epsilon'/\\epsilon_0\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("x axis scale -3 4\nw axis scale -3 4 suppress\n");
        printf("y axis scale 0 15\nz axis scale 0 15 suppress\n");
        printf("plot line cross green marker 0.08\n");                      20
        printf("input \"%s.e1\"\n", argv[1]);

        printf("window bottom\n");
        printf("label left \"log(\\epsilon'')/\\epsilon_0)\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("x axis scale -3 4\nw axis scale -3 4 suppress\n");
        printf("y axis scale -3 3\nz axis scale -3 3 suppress\n");
        printf("plot line cross green marker 0.08\n");
        printf("input \"%s.e2\"\n", argv[1]);
}                                                                           30
```

eplot3x.c

```
/* This program when piped into plop plots the .e1 and .e2 files
   of the three wavelengths of the 3-l sensor. The files
   name.1.e1      name.2.e1      name.3.e1
   name.1.e2      name.2.e2      name.3.e2
   must exist in that directory. It differs from eplot3 in that it adds
   axis scale information and makes output uniform.

   Yanko Sheiretov      11/17/93                                    */

#include <stdio.h>                                                    10
main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 2) {
                fprintf(stderr, "Usage:  \teplot3x <name>\n");
                exit();}
        printf("window top\n");
        printf("title top \"Complex \\epsilon with 3-\\lambda.  File:  %s\"\n",
                                                          argv[1]);   20
        printf("label top \"\\triangle:  \\lambda = 5 mm   \\square:");
        printf(" \\lambda = 2.5 mm   \\circle:  \\lambda = 1 mm\"\n");
        printf("label left \"\\epsilon'/\\epsilon_0\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("x axis scale -3 4\nw axis scale -3 4 suppress\n");
        printf("y axis scale 0 7.5\nz axis scale 0 7.5 suppress\n");
        printf("plot triangles green marker 0.08\n");
        printf("input \"%s.1.e1\"\n", argv[1]);
        printf("plot squares green marker 0.08\n");
        printf("input \"%s.2.e1\"\n", argv[1]);                       30
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.3.e1\"\n", argv[1]);

        printf("window bottom\n");
```

```
        printf("label left \"log(\\epsilon'')/\\epsilon_0)\"\n");
        printf("label bottom \"log(freq)\"\n");
        printf("x axis scale -3 4\nw axis scale -3 4 suppress\n");
        printf("y axis scale -3 3\nz axis scale -3 3 suppress\n");
        printf("plot triangles green marker 0.08\n");
        printf("input \"%s.1.e2\"\n", argv[1]);                    40
        printf("plot squares green marker 0.08\n");
        printf("input \"%s.2.e2\"\n", argv[1]);
        printf("plot circles green marker 0.08\n");
        printf("input \"%s.3.e2\"\n", argv[1]);
}
```

```
# include       <stdio.h>
# define        EOL     '\n'            /* end of line definition */
# define        MAXPTS 500

/* This program reads a file generated by the controller box and outputs
   to stdout plotting commands to plot the raw data.
        Yanko Sheiretov         9/11/92         Revised:  8/20/93       */

main(argc, argv)
int     argc;                                                           10
char    *argv[];

{
        char    junk;
        FILE    *fpses;
        double  ch, f[MAXPTS], g[MAXPTS], p[MAXPTS], o[MAXPTS];
        int year, month, day, hour, minute;
        int n, i;

        if (argc != 2) {                                                20
                fprintf(stderr, "Usage:  \traw <file> | plop\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {
            fprintf(stderr, "Error:  cannot open %s\n",argv[1]);
            exit(1);
            }

        while((junk=fgetc(fpses)!=','));                                30
        fscanf(fpses,"%lf,%d,%d,%d,%d,%d",&ch,&year,&month,&day,&hour,&minute);
        while((junk=fgetc(fpses)) != EOL && junk != EOF);
        n = 0;
        while(1) {
```

```
            do junk=fgetc(fpses);
        while (junk != EOF && junk != EOL && junk != ']' && junk != ',');
            if (junk == EOF) break;
            if (junk == EOL || junk == ']') continue;
            fscanf(fpses,"%lf,%lf,%lf,%lf,%lf",&ch,f+n,g+n,p+n,o+n);
            n++;                                                          40
            while(fgetc(fpses) != ']');
            }
    fclose(fpses);

    printf("window vertical 1/3\ntitle top \"Raw Data\"\n");
    printf("label top \"date:  %d/%d/%d \"\n", month, day, year);
    printf("text over 0.1 right 0.3 size 0.12 \"file:  %s\"\n", argv[1]);
    printf("label bottom \"log(freq)\"\n");
    printf("label left \"Gain [dB]\"\nplot green cross marker 0.08\n");
    for(i=0;i<n;i++) printf("%f\t%f\n",f[i],g[i]);                         50

    printf("window vertical 1/3\n");
    printf("label bottom \"log(freq)\"\n");
    printf("label left \"Phase [deg]\"\nplot green cross marker 0.08\n");
    for(i=0;i<n;i++) printf("%f\t%f\n",f[i],p[i]);

    printf("window vertical 1/3\n");
    printf("label bottom \"log(freq)\"\n");
    printf("label left \"Offset [mV]\"\nplot green cross marker 0.08\n");
    for(i=0;i<n;i++) printf("%f\t%f\n",f[i],1000.*o[i]);                   60

    fprintf(stderr,"Done ...  %d data points.\n",n);
    fflush(stdout);
}
```

```
# include       <stdio.h>
# define        EOL     '\n'            /* end of line definition */
# define        MAXPTS 500

/* This program reads a file generated by the controller box and outputs
   to stdout plotting commands to plot the raw data for three-wavelength
   sensors.

        Yanko Sheiretov         8/31/92         R. 12/17/93             */

main(argc, argv)
int     argc;
char    *argv[];

{
        char    junk;
        FILE    *fpses;
        double  temp, f[MAXPTS][3], g[MAXPTS][3], p[MAXPTS][3], o[MAXPTS][3];
        int     year, month, day, hour, minute, channel;
        double  ff, gg, pp, oo;
        int     n, i;

        if (argc != 2) {
                fprintf(stderr, "Usage:  \traw3 <file> | plop\n");
                exit(); }


        if ((fpses = fopen(argv[1],"r")) == NULL) {
            fprintf(stderr, "Error:  cannot open %s\n",argv[1]);
            exit(1);
            }

        while((junk=fgetc(fpses)!=','));
        fscanf(fpses,"%lf,%d,%d,%d,%d,%d",
```

```
        &temp, &year, &month, &day, &hour, &minute);
while((junk=fgetc(fpses)) != EOL && junk != EOF);
n = -1;
while(1) {
        do junk=fgetc(fpses);
    while (junk != EOF && junk != EOL && junk != ']' && junk != ',');
        if (junk == EOF) break;
        if (junk == EOL || junk == ']') continue;
        fscanf(fpses,"%d,%lf,%lf,%lf,%lf",
                &channel, &ff, &gg, &pp, &oo);
        if (channel == 1) n++;
        if (channel > 0 && channel < 4) {
                f[n][channel-1] = ff;
                g[n][channel-1] = gg;
                p[n][channel-1] = pp;
                o[n][channel-1] = oo; }
        else {
                fprintf(stderr, "Bad channel number:  %d\n", channel);
                exit(); }
        while(fgetc(fpses) != ']');
        }
fclose(fpses);

printf("window vertical 1/3\ntitle top \"3-\\lambda Raw Data:");
printf(" %d/%d/%d \"\n", month, day, year);
printf("label top \"\\triangle:  \\lambda = 5 mm   \\square:");
printf(" \\lambda = 2.5 mm   \\circle:  \\lambda = 1 mm\"\n");
printf("text over 0.1 right 0.3 size 0.12 \"file:  %s\"\n", argv[1]);
printf("label bottom \"log(freq)\"\n");
printf("label left \"Gain [dB]\"\nplot green triangles marker 0.08\n");
for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][0], g[i][0]);
printf("plot green squares marker 0.08\n");
for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][1], g[i][1]);
printf("plot green circles marker 0.08\n");
for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][2], g[i][2]);
```

```
        printf("window vertical 1/3\n");
        printf("label bottom \"log(freq)\"\n");
        printf("label left \"Phase [deg]\"\n");
        printf("plot green triangles marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][0], p[i][0]);
        printf("plot green squares marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][1], p[i][1]);
        printf("plot green circles marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][2], p[i][2]);
                                                                        80
        printf("window vertical 1/3\n");
        printf("label bottom \"log(freq)\"\n");
        printf("label left \"Offset [mV]\"\n");
        printf("plot green triangles marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][0], 1000.*o[i][0]);
        printf("plot green squares marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][1], 1000.*o[i][1]);
        printf("plot green circles marker 0.08\n");
        for(i=0;i<=n;i++) printf("%f\t%f\n", f[i][2], 1000.*o[i][2]);
                                                                        90
        fprintf(stderr,"Done ...  %d distinct frequencies.\n", n+1);
        fflush(stdout);
}
```

Appendix H

Program Listings for the Parameter Estimation Algorithms

H.1 Description

This appendix contains the code for all of the parameter estimation routines described in Chapter 4. Table H.1 lists the function of the code in every file. The rest of this section contains the description of the different programs and function. An attempt is made to organize this list by level of abstraction, beginning at the lowest level. Figure H-1 shows a diagram of the interdependence of the different files.

The routine **gp.c** gives the solution of the *forward* problem, i.e. given the properties of the sensor and the layer structure above it, it computes the gain and the phase of the response. This routine calls the functions in **scap.c, coef.c, solve.c,** and **admit.c**. All of the subsidiary routines mentioned in this paragraph were written in almost direct translation from Dr. M. Zaretsky's FORTRAN code [3].

The multidimensional parameter estimation routines use **gp.c** in some sort of an iterative fashion in order to solve the *inverse* problem, i.e. finding out something about the materials above the sensors, given the measured gain and phase data. For a full description of all programs in this appendix, read Chapter 4.

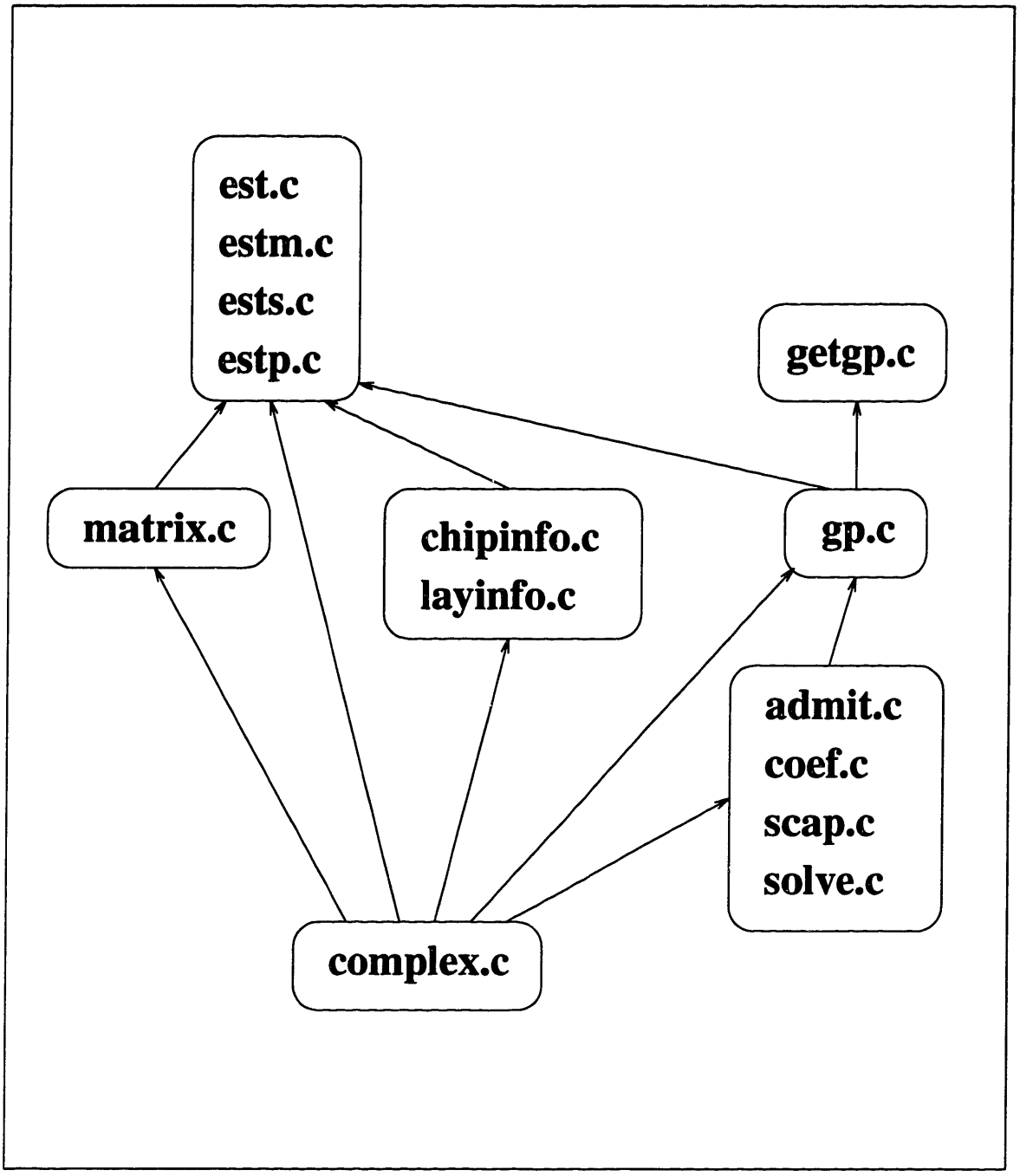

Figure H-1: Interdependence of parameter estimation routines

Name	Description
Multidimensional Parameter Estimation	
est.c	Root finding
estm.c	Minimization based on Powell's method [9]
ests.c	Minimization based on the downhill simplex method [9]
estp.c	Root finding with assumed profile function
getgp.c	Finds the gain and phase given a layer structure
Subsidiary Parameter Estimation Routines	
scap.c	Calculates surface capacitance density
coef.c	Calculates the coefficients of the collocation point matrix
solve.c	Solves the collocation point matrix
admit.c	Calculates the impedances in the sensor model
gp.c	Uses all of the above to calculate gain and phase
testgp.c	tests **gp.p**
Tools	
complex.c	Defines functions operating on complex numbers
matrix.c	Functions on matrices of complex numbers
Input/Output	
chipinfo.c	Sensor information
layinfo.c	Layer information

Table H.1: Summary of parameter estimation routines

H.2 Makefile

makefile

```
# makefile for estimation routines
# Yanko Sheiretov          1/24/94

.SUFFIXES: .c .h

CC = cc -c

CFLAGS =

LINK = cc -o                                                    10

.c.o:
        $(CC) $(CFLAGS) $<

est: est.o matrix.o chipinfo.o layinfo.o gp.o admit.o \
        coef.o complex.o solve.o scap.o
        $(LINK) est est.o matrix.o chipinfo.o layinfo.o \
        gp.o admit.o coef.o complex.o solve.o scap.o -lm

estp: estp.o matrix.o chipinfo.o layinfo.o gp.o admit.o \       20
        coef.o complex.o solve.o scap.o slope.o
        $(LINK) estp estp.o matrix.o chipinfo.o layinfo.o \
        gp.o admit.o coef.o complex.o solve.o scap.o slope.o -lm

estm: estm.o chipinfo.o layinfo.o gp.o admit.o \
        coef.o complex.o solve.o scap.o
        $(LINK) estm estm.o chipinfo.o layinfo.o \
        gp.o admit.o coef.o complex.o solve.o scap.o -lm

ests: ests.o chipinfo.o layinfo.o gp.o admit.o \                30
        coef.o complex.o solve.o scap.o
        $(LINK) ests ests.o chipinfo.o layinfo.o \
```

```
        gp.o admit.o coef.o complex.o solve.o scap.o -lm

getgp: getgp.o chipinfo.o layinfo.o gp.o admit.o \
        coef.o complex.o solve.o scap.o
        $(LINK) getgp getgp.o chipinfo.o layinfo.o gp.o \
        admit.o coef.o complex.o solve.o scap.o -lm

test: test.o gp.o admit.o coef.o complex.o solve.o scap.o                    40
        $(LINK) test test.o \
        gp.o admit.o coef.o complex.o solve.o scap.o -lm

admit.o coef.o complex.o solve.o scap.o: est.h complex.h

gp.o: est.h complex.h objects.h

est.o estp.o: matrix.h objects.h complex.h

layinfo.o chipinfo.o estm.o ests.o: objects.h complex.h                      50

getgp.o test.o: objects.h complex.h
```

H.3 Header Files

complex.h

```
/* Header file for operations with complex numbers.
   Yanko Sheiretov          1/20/94          1/25/94                    */

typedef struct {
        double x,y;
        } complex;

#define re(z)           ((z).x)
#define im(z)           ((z).y)

                                                                        10

complex cmplx(), recip(), plus(), times(), scale(), minus(), over(), csq();
double ccabs();
```

est.h

```
/* Header file for files used in the estimation routines.
   Yanko Sheiretov          1/12/94          1/25/94                    */

#include <math.h>
#include <stdio.h>
#include "complex.h"
#include "objects.h"
#define PI              3.141592654
#define N2              200     /* maximum number of fourier summation terms */
#define N1              50      /* maximum number of collocation points */
#define MF              N2+1    /* array dimension: indexing begins at 1 */
#define grid(x)         (gr[(x)])

extern  double  lambda;
extern  double  eox;    /* substrate permittivity */
extern  double  h;      /* thickness */
extern  double  g;      /* interelectrode thickness */
extern  double  yload;  /* load impedance */
extern  complex ckmin;
extern  complex rea, rsea;
extern  int     k, N, nmax, num;
extern  double  en0;
extern  double  wl;     /* frequency */
extern  double  ap, bp;
extern  complex y11, y12, yp;
extern  double  gr[N1+2];       /* grid points */
void    coef(), scap(), admit(), solve(), gridinit();
complex fhsum();
double  fl(), fsum(), fact();
```

```
void ysolve(), invert(), mul(), mulv();
```

objects.h

```
/* This header file defines structures, etc. which are useful for the
   communications between an estimation algorithm and the forward function.
   Yanko Sheiretov          1/20/94                                          */

/* describes a homogeneous layer above the electrodes */
struct layer {
        complex bulk;
        complex surface;
        double  thickness;
        };                                                                   10
#define ML      20
#define sq(x)           ((x)*(x))       /* a useful macro */

struct sensor {
        int     k;      /* number of collocation points */
        int     N;      /* number of Fourier terms */
        double  lambda; /* spatial wavelength */
        double  eox;    /* substrate permittivity */
        double  h;      /* substrate thickness */
        double  g;      /* interelectrode thickness */                       20
        double  yload;  /* load impedance */
        double  en0;    /* imposed electric field */
        double  w1;     /* AC frequency */
        double  ap;     /* parasitic coefficient ap */
        double  bp;     /* parasitic coefficient bp */
        complex ea;     /* complex bulk permittivity above electrodes */
        complex sea;    /* complex surface permittivity above electrodes */
        };

complex gp();                                                                30
```

H.4 Main Parameter Estimation Routines

est.c

```
/* This is the main file which contains the control function for the
   multidimensional parameter estimation routine. It used a form of
   the Secant method to find out the bulk properties of a
   set of unknown layers. Data is taken from several sensors with
   different spatial wavelengths.

   Yanko Sheiretov          2/5/94          3/9/94                  */

/* Major revision: the analytical properties of the complex gain as a
   function of the complex permittivities is taken into account.

   Yanko Sheiretov          3/24/94         5/3/94                  */

#include "complex.h"
#include "objects.h"
#include "matrix.h"
#include <stdio.h>
#include <math.h>

void chipinfo(), layinfo(), chipinfoout(), layinfoout();

#define PI      3.141592654
#define TOLR    0.1             /* result tolerance factor */
#define TOLV    0.5e-12         /* independent vector tolerance */
#define INCA    1.0e-13         /* additive increment for derivatives */
#define INCM    0.01            /* multiplicative increment for derivatives */
#define EMIN    8.854e-12       /* minimal permittivity */
#define SMIN    -1e-12          /* minimal conductivity */

#define MAXK 25         /* number of collocation points */
#define MAXN 100        /* number of Fourier terms */
#define MAXITER 25      /* maximum number of iterations */
```

```
#define MAXDAMP 6    /* maximum number of damping steps */
#define NEWJAC 1     /* how often a new jacobian is calculated */

main()
{
        double  w1;     /* AC frequency */
        complex ea;     /* complex bulk permittivity above electrodes */
        complex sea;    /* complex surface permittivity above electrodes */
                        /* The above two quantities are assumes to be
                                the same for all sensors */
        char    name[20];       /* output file name */
        FILE    *fp;            /* output file pointer */
        int     n;              /* number of layers */
        int     df;             /* number of degrees of freedom, i.e.
                                the number of sensors and unknown layers */
        int     u[ML];          /* index numbers for unknown layers */
        int     iter;           /* number of iterations */
        complex jac[ML][ML];    /* jacobian */
        complex ijac[ML][ML];   /* inverse jacobian */
        complex old[ML];        /* old guess vector */
        complex new[ML];        /* new guess vector */
        complex res[ML];        /* result vector */
        complex meas[ML];       /* measured values vector */
        complex r;              /* result register */
        struct  sensor s[ML];   /* array of sensors */
        struct  layer l[ML];    /* array of layers */
        int     i, m;           /* counters */
        double  sum=0.0, newsum, oldsum=0.0, sum1, sum2;
        complex dx;             /* derivative increment */
        int     damp=0;         /* number of damping steps */
        char    newjac=1;       /* flags whether a new jacobian is needed */
        char    isbad();        /* tests the validity of the new point */
        int     last=-1;        /* iteration number for the last time a new
                                        jacobian was calculated */
        complex to_e_anal(), to_e_stan(), to_g_anal();
        char    val[5];         /* whether parameter validity is done */
```

```
/************** Get information from user ****************************/

fputs("Please enter output file name:  ", stdout);
scanf("%s", name);
if((fp = fopen(name, "w")) == NULL) {
        fprintf(stderr, "est error:  can't open %s\n", name);
        exit(); }
fputs("Please enter the AC frequency [Hz]:  ", stdout);
scanf("%lf", &w1);
w1 *= 2.0*PI;     /* convert to rad/s */
fputs("Please enter the bulk permittivity above the ", stdout);
puts("electrodes [F/m],[S/m]:");
scanf("%lf,%lf", &re(ea), &im(ea));
fputs("Please enter the surface permittivity above the ", stdout);
puts("electrodes [F],[S]:");
scanf("%lf,%lf", &re(sea), &im(sea));
fputs("\nPlease enter the number of layers:  ", stdout);
scanf("%d", &n);
fputs("Please enter the number of sensors (unknown layers):  ", stdout);
scanf("%d", &df);
puts("Please enter the index numbers of the unknown layers,");
puts("separated with commas, starting from the infinite half");
puts("space (number 0), e.g.  2,3,5");
for(i=0; i<df; i++) {
        scanf("%d", u+i);
        if (i < df-1) while (getchar() != ','); }
putchar('\n');

for(i=0; i<df; i++) {
        printf("Please enter information about sensor number %d:\n",i);
        chipinfo(s+i); }
puts("\nNow enter information about the layers.  The values entered");
puts("for the bulk properties of the unknown layers will be used as");
puts("the initial guess.  Layer number 0 is the topmost layer,");
puts("i.e the infinite half space.");
```

```
        for(i=0; i<n; i++) {
                printf("Please enter information about layer number %d:\n", i);
                layinfo(l+i); }

        puts("\nPlease enter the measured gain and phase for :  ([dB],[deg])");
        for(i=0; i<df; i++) {                                                    110
                printf("\tsensor number %d :  ", i);
                scanf("%lf,%lf", &re(meas[i]), &im(meas[i])); }
        fputs("\nWould you like to perform parameter range damping?  ", stdout);
        scanf("%3s%*s", val);
        *val = *val=='y'|| *val=='Y' ? 1 : 0;
        puts("\nThe process now begins ...");

/******************* Print out input data ********************************/

        fprintf(fp, "Output file name:  %s\n", name);                            120
        fprintf(fp, "Number of collocation points:  %d\n", MAXK);
        fprintf(fp, "Number of Fourier terms:  %d\n", MAXN);
        fprintf(fp, "Maximum number of iterations:  %d\n", MAXITER);
        fprintf(fp, "Maximum number of damping cycles:  %d\n", MAXDAMP);
        fprintf(fp, "AC frequency [Hz]:  %g\n", w1/PI/2.0);
        fprintf(fp,
                "Bulk permittivity above the electrodes [F/m],[S/m]:  %g,%g\n",
                re(ea), im(ea));
        fprintf(fp,
                "Surface permittivity above the electrodes [F],[S]: %g,%g\n",    130
                re(sea), im(sea));
        fprintf(fp, "Number of layers:  %d\n", n);
        fprintf(fp, "Number of sensors:  %d\n", df);
        fputs("Index numbers of the unknown layers:  ", fp);
        for(i=0;i<df;i++) {
                if (i<df-1) fprintf(fp, "%d,  ", u[i]);
                else fprintf(fp, "%d\n", u[i]); }
        for(i=0; i<df; i++) {
                fprintf(fp, "Information about sensor number %d:\n", i);
                chipinfoout(s+i, fp); }                                          140
```

```
        for(i=0; i<n; i++) {
                fprintf(fp, "Information about layer number %d:\n", i);
                layinfoout(l+i, fp); }

        fputs("Measured gain and phase for :  ([dB],[deg])\n", fp);
        for(i=0; i<df; i++) fprintf(fp, "\tsensor number %d :  %g,%g\n", i,
                                        re(meas[i]), im(meas[i]));
        fprintf(fp, "\nParameter range damping %s.\n", *val ? "ON" : "OFF");
        fflush(fp);

/******************  Fill in remaining sensor data **********************/

        for(i=0; i<df; i++) {
                (s[i]).k = MAXK;
                (s[i]).N = MAXN;
                (s[i]).en0 = 0.0;           /* No superimposed field assumed */
                (s[i]).w1 = w1;
                (s[i]).ea = ea;
                (s[i]).sea = sea; }

/************** Convert meas to analytical form **************************/

        for(i=0; i<df; i++) meas[i] = to_g_anal(meas[i]);

/******************  Iteration process  ******************************/

        /* initialize first guess */
        for(i=0; i<df; i++) new[i] = to_e_anal((l[u[i]]).bulk, w1);

        /* iteration control loop */
        for(iter=1; iter <= MAXITER; iter++) {
                printf("Iteration number %d\n", iter);
                for (i=0;i<df;i++) printf("new[%d] = (%g, %g)\n", i,
                        re(new[i]), im(new[i]));

                if (newjac && last == iter-1) {
```

```
        printf("Convergence failure:  Badness %g > %g\n",
                oldsum, TOLR);
        fprintf(fp, "Convergence failure:  Badness %g > %g\n",
                oldsum, TOLR);                                          180
        exit(0);
        }

/* calculate res */
for(m=0; m<df; m++) {
        /* place guess into layer structures */
        if (iter > 1)
           for(i=0; i<df; i++)
                (l[u[i]]).bulk = to_e_stan(new[i], w1);
        res[m] = to_g_anal(gp(n, l, s+m));                              190
        printf("res[%d] = (%g, %g)\n", m,
                re(res[m]), im(res[m]));
        }

/* test to see whether res is close to meas */
for(sum1=0.0, sum2=0.0, i=0; i<df; i++) {
        newsum = ccabs(minus(res[i], meas[i]));
        sum1 += sq(newsum);
        newsum = ccabs(plus(res[i], meas[i]));
        sum2 += sq(newsum);                                             200
        }
newsum = sqrt(sum1/sum2);
printf("newsum = %g\n", newsum);

if(newsum < TOLR) {
        puts("res is close enough to meas:");
        printf("\tnewsum = %g < %g\n", newsum, TOLR);
        break; }

/* test to see if damping is needed */                                  210
if (newsum > oldsum && !newjac) {
        printf ("Damping needed:  step #%d\n", damp+1);
```

```
            printf ("because %g > %g\n", newsum, oldsum);
            iter--;  /* doesn't count as an iteration */
            if (damp >= MAXDAMP) { /* new jacobian is needed */
                    puts("Even worse -- new jacobian!");
                    newjac = 1;
                    for (i=0; i<df; i++) new[i] = old[i];
                    continue;
                    }                                                    220
            else {
                    damp++;
                    for (i=0; i<df; i++)
                       new[i] = scale(plus(new[i], old[i]), 0.5);
                    continue;
                    }
            }
    else {
            oldsum = newsum;
            damp = 0;                                                    230
            }

    /* calculate jacobian partial derivatives */
    if (iter-last >= NEWJAC) newjac = 1;
    if (newjac) puts("\nJacobian calculation !");
    if (newjac) for(m=0; m<df; m++) {
            if (ccabs(new[m]) != 0.0) dx = scale(new[m], INCM);
            else dx = cmplx(INCA, 0.0);
            (l[u[m]]).bulk = to_e_stan(plus(new[m], dx), w1);
            printf("new[%d] is now (%g, %g)\n", m, re(new[m]) +          240
                    re(dx), im(new[m]) + im(dx));
            for(i=0; i<df; i++) {
                    r = to_g_anal(gp(n, l, s+i));
                    jac[i][m] = over(minus(r, res[i]), dx);
                    printf("r[%d] = (%g, %g)\n", i, re(r), im(r));
                    printf("jac[%d][%d] = (%g, %g)\n", i, m,
                            re(jac[i][m]), im(jac[i][m]));
                    }
```

```
        (l[u[m]]).bulk = to_e_stan(new[m], w1);        /* restore */
        }                                                              250

/* new becomes old */
/* subtract meas from res */
for(i=0; i<df; i++) {
        old[i] = new[i];
        res[i] = minus(res[i], meas[i]);
        }

/* calculate correction vector */
if (newjac) invert(jac, ijac, df);                                     260
mulv(ijac, res, new, df);
if (newjac) last = iter;
newjac = 0;

/* find new values for new */
for(i=0; i<df; i++) new[i] = minus(old[i], new[i]);

/* damping caused by parameters out of range */
if (*val) {
 for (m=0; m < MAXDAMP && isbad(new, df); m++) {                       270
  for(i=0; i<df; i++)
        new[i] = scale(plus(new[i], old[i]), 0.5);
  printf("Parameter range damping:  #%d\n", m+1);
  for(i=0; i<df; i++) printf("%g, %g, ", re(new[i]),
                        im(new[i]));
  putchar('\n');
  }
 if (m == MAXDAMP) {
        if (last == iter) {
          puts("Failure due to parameters being out of range");        280
        fputs("Failure due to parameters being out of range\n",
                        fp);
          exit(0);
                }
```

```
                newjac = 1;
                iter--;
                for (i=0; i<df; i++) new[i] = old[i];
                puts("New jacobian due to parameters out of range.");
                continue;
                }                                                    290
        }

        /* test to see if new is close to old */
        for(sum=0.0,i=0; i<df; i++)
                sum += ccabs(minus(new[i], old[i]));
        if(sum < TOLV) {
                puts("new is close enough to old:");
                printf("\tsum = %g < %g\n", sum, TOLV);
                break; }
                                                                     300
        /* End of iteration cycle */
        fflush(stdout);
        }

/****************** Print Out Results ********************************/

    if (iter > MAXITER) {
            puts("Problems with convergence.");
            fputs("Problems with convergence.\n", fp);
            }                                                        310
    else {
            /* place results into layer structures */
            for(i=0; i<df; i++) (l[u[i]]).bulk = to_e_stan(new[i], w1);

            puts("Done -- see output file for results.");
            fprintf(fp, "\n%15s%15s%15s\n", "Layer number",
                            "Permittivity", "Conductivity");
            for(i=0; i<46; i++) putc('-', fp);
            fputs("\n$\n", fp);
            for (i=0; i<df; i++)                                     320
```

```
                        fprintf(fp, "%15d%15e%15e\n", u[i], re((l[u[i]]).bulk),
                                                    im((l[u[i]]).bulk));
                fputs("$\n", fp); }

        fclose(fp);
        fflush(stdout);
}

/* The following function tests whether the values in the array new[] are
   valid. Returns a boolean value */                                        330

char    isbad(new, n)
        complex *new;
        int n;
{
        char r=0;
        int i;

        for(i=0; i<n; i++) {
                r = r || re(new[i]) < EMIN || -im(new[i]) < SMIN;           340
                if (r) break;
                }
        return (r);
}


/* The following functions convert numbers for the complex permittivity
   and the complex gain between analytical and standard representations:

   (g', g")  <-->  (M, phi)          (e, s)  <-->  (e', -e")                350

   M = sqrt(g'*g' + g"*g")           e' = e
   phi = arctan (g"/g')              -e" = -s/w1
                                                                    */

complex to_e_anal(z, w)
```

```
        complex z;
        double  w;
{
        return (cmplx(re(z), -im(z)/w));                                    360
}

complex to_e_stan(z, w)
        complex z;
        double  w;
{
        return (cmplx(re(z), -w*im(z)));
}

complex to_g_anal(z)                                                        370
        complex z;
{
        double r, theta;

        r = pow(10.0, re(z)/20.0);
        theta = im(z)*PI/180.0;

        return(cmplx(r*cos(theta), r*sin(theta)));
}
                                                                            380
```

```
/* This version of the parameter estimation program "est.c" uses a different
   approach: instead of searching for a root, we search for a minimum of an
   error function (func()). This has the advantage that the numerical method
   is more stable. Also, it allows for having more sensors than unknown layers
   and does not fail if due to experimental errors no roots exist. Preliminary
   tests seem to indicate that it takes more computation time than "est.c".

   Yanko Sheiretov          3/28/94                                        */

#include "complex.h"                                                      10
#include "objects.h"
#include <stdio.h>
#include <math.h>

void chipinfo(), layinfo(), chipinfoout(), layinfoout();
double pcom[ML],xicom[ML];
double sqrarg;
int df,sen;

#define FTOL   0.001     /* Fractional tolerance for the error function */ 20
#define ATOL   0.01      /* Absolute tolerance for the error function */
#define UV     1e-13     /* Magnitude of search unit vectors */
#define TOL    0.05      /* Tolerance for the one-dimensional minimization */
#define CGOLD 0.3819660  /* 1 - phi (Golden ratio) */
#define GOLD 1.618034    /* 1 + phi (Golden ratio) */
#define GLIMIT 100.0     /* limit for parabolic extrapolation */
#define TINY 1.0e-20     /* a tiny number used to prevent division by zero */

#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)                                 30
#define SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

#define PI     3.141592654
```

```
#define MAXK 25        /* number of collocation points */
#define MAXN 100       /* number of Fourier terms */
#define MAXITER 50     /* maximum number of iterations */

double  w1;            /* AC frequency */                              40
complex ea;            /* complex bulk permittivity above electrodes */
complex sea;           /* complex surface permittivity above electrodes */
                       /* The above two quantities are assumes to be
                                  the same for all sensors */
char    name[20];      /* output file name */
FILE    *fpout;        /* output file pointer */
int     n;             /* number of layers */
int     u[ML];         /* index numbers for unknown layers */
complex meas[ML];      /* measured values vector */
double  res[2*ML];     /* result register */                           50
struct  sensor s[ML];  /* array of sensors */
struct  layer l[ML];   /* array of layers */

main()
{
        int i;
        double min, powell();

/**************  Get information from user  ***************************/
                                                                        60
        fputs("Please enter output file name:  ", stdout);
        scanf("%s", name);
        if((fpout = fopen(name, "w")) == NULL) {
                fprintf(stderr, "est error:  can't open %s\n", name);
                exit(); }
        fputs("Please enter the AC frequency [Hz]:  ", stdout);
        scanf("%lf", &w1);
        w1 *= 2.0*PI;     /* convert to rad/s */
        fputs("Please enter the bulk permittivity above the ", stdout);
        puts("electrodes [F/m],[S/m]:");                                70
```

```
scanf("%lf,%lf", &re(ea), &im(ea));
fputs("Please enter the surface permittivity above the ", stdout);
puts("electrodes [F],[S]:");
scanf("%lf,%lf", &re(sea), &im(sea));
fputs("\nPlease enter the number of layers:  ", stdout);
scanf("%d", &n);
fputs("Please enter the number of sensors:  ", stdout);
scanf("%d", &sen);
puts("Please enter the number of unknown layers.  Should be less than");
fputs("or equal to the number of sensors:  ", stdout);                    80
scanf("%d", &df);
df *= 2;
puts("Please enter the index numbers of the unknown layers,");
puts("separated with commas, starting from the infinite half");
puts("space (number 0), e.g.  2,3,5");
for(i=0; i<df/2; i++) {
        scanf("%d", u+i);
        if (i < df/2-1) while (getchar() != ',');  }
putchar('\n');
                                                                          90
for(i=0; i<sen; i++) {
        printf("Please enter information about sensor number %d:\n",i);
        chipinfo(s+i);  }
puts("\nNow enter information about the layers.  The values entered");
puts("for the bulk properties of the unknown layers will be used as");
puts("the initial guess.  Layer number 0 is the topmost layer,");
puts("i.e the infinite half space.");
for(i=0; i<n; i++) {
        printf("Please enter information about layer number %d:\n", i);
        layinfo(l+i);  }                                                  100

puts("\nPlease enter the measured gain and phase for :  ([dB],[deg])");
for(i=0; i<sen; i++) {
        printf("\tsensor number %d :  ", i);
        scanf("%lf,%lf", &re(meas[i]), &im(meas[i]));  }
puts("\nThe process now begins ...");
```

```
/******************** Print out input data *********************************/

        fprintf(fpout, "Output file name:  %s\n", name);                        110
        fprintf(fpout, "Number of collocation points:  %d\n", MAXK);
        fprintf(fpout, "Number of Fourier terms:  %d\n", MAXN);
        fprintf(fpout, "Maximum number of iterations:  %d\n", MAXITER);
        fprintf(fpout, "AC frequency [Hz]:  %g\n", w1/PI/2.0);
        fprintf(fpout,
                "Bulk permittivity above the electrodes [F/m],[S/m]:  %g,%g\n",
                re(ea), im(ea));
        fprintf(fpout,
                "Surface permittivity above the electrodes [F],[S]: %g,%g\n",
                re(sea), im(sea));                                              120
        fprintf(fpout, "Number of layers:  %d\n", n);
        fprintf(fpout, "Number of sensors:  %d\n", sen);
        fprintf(fpout, "number of unknown layers:  %d\n", df/2);
        fputs("Index numbers of the unknown layers:  ", fpout);
        for(i=0;i<df/2;i++) {
                if (i<df/2-1) fprintf(fpout, "%d, ", u[i]);
                else fprintf(fpout, "%d\n", u[i]); }
        for(i=0; i<sen; i++) {
                fprintf(fpout, "Information about sensor number %d:\n", i);
                chipinfoout(s+i, fpout); }                                      130
        for(i=0; i<n; i++) {
                fprintf(fpout, "Information about layer number %d:\n", i);
                layinfoout(l+i, fpout); }

        fputs("Measured gain and phase for :  ([dB],[deg])\n", fpout);
        for(i=0; i<sen; i++) fprintf(fpout, "\tsensor number %d :  %g,%g\n", i,
                                        re(meas[i]), im(meas[i]));
        fflush(fpout);

/******************  Fill in remaining sensor data **********************/      140

        for(i=0; i<sen; i++) {
```

```
            (s[i]).k = MAXK;
            (s[i]).N = MAXN;
            (s[i]).en0 = 0.0;          /* No superimposed field assumed */
            (s[i]).w1 = w1;
            (s[i]).ea = ea;
            (s[i]).sea = sea; }

        /* initialize first guess */                                          150
        for(i=0; i<df; i++) res[i] = i%2==0 ? re((l[u[i/2]]).bulk) :
                                             im((l[u[i/2]]).bulk);
        min = powell(res);

/******************** Print Out Results ***********************************/

        /* place results into layer structures */
        for(i=0; i<df/2; i++) (l[u[i]]).bulk = cmplx(res[2*i], res[2*i+1]);

        puts("Done -- see output file for results.");                     160
        printf("Minimum achieved:  %g\n\n", min);
        fprintf(fpout, "Minimum achieved:  %g\n\n", min);
        fprintf(fpout, "\n%15s%15s%15s\n", "Layer number",
                        "Permittivity", "Conductivity");
        for(i=0; i<46; i++) putc('-', fpout);
        fputs("\n$\n", fpout);
        for (i=0; i<df/2; i++)
                fprintf(fpout, "%15d%15e%15e\n", u[i], re((l[u[i]]).bulk),
                                             im((l[u[i]]).bulk));
        fputs("$\n", fpout);                                              170

        fclose(fpout);
        fflush(stdout);
}

double powell(p)
double p[];
{
```

```
int i,ibig,j,iter;
double t,fptt,fp,del,fret;                                          180
double pt[ML],ptt[ML],xit[ML],xi[ML][ML];
void linmin();
double func();

fret=func(p);
for (j=0;j<df;j++) pt[j]=p[j];
for (iter=1;;iter++) {
        printf("\tPowell Iteration #%d\n", iter);
        fp=fret;
        ibig=0;                                                     190
        del=0.0;
        for (i=0;i<df;i++) {
                for (j=0;j<df;j++) xit[j]=xi[j][i]=j==i?UV:0.0;
                fptt=fret;
                linmin(p,xit,&fret);
                if (fabs(fptt-fret) > del) {
                        del=fabs(fptt-fret);
                        ibig=i;
                        }
                }                                                   200
        if (2.0*fabs(fp-fret) <= FTOL*(fabs(fp)+fabs(fret)) ||
                fabs(fret) < ATOL) return (fret);
        if (iter == MAXITER) {
                puts("Powell - Problems with convergence.");
                exit(0);
                }
        for (j=0;j<df;j++) {
                ptt[j]=2.0*p[j]-pt[j];
                xit[j]=p[j]-pt[j];
                pt[j]=p[j];                                         210
                }
        fptt=func(ptt);
        if (fptt < fp) {
                t=2.0*(fp-2.0*fret+fptt)*SQR(fp-fret-del)-
```

```
                        del*SQR(fp-fptt);
                    if (t < 0.0) {
                        linmin(p,xit,&fret);
                        for (j=0;j<df;j++) xi[j][ibig]=xit[j];
                    }
                }                                                    220
            }
        }

        void linmin(p,xi,fret)
        double p[],xi[],*fret;
        {
            int j;
            double xx,xmin,fx,fb,fa,bx,ax;
            double brent(),f1dim();
            void mnbrak();                                           230

            for (j=0;j<df;j++) {
                pcom[j]=p[j];
                xicom[j]=xi[j];
            }
            ax=0.0;
            xx=1.0;
            bx=2.0;
            mnbrak(&ax,&xx,&bx,&fa,&fx,&fb);
            *fret=brent(ax,xx,bx,&xmin);                             240
            for (j=0;j<df;j++) {
                xi[j] *= xmin;
                p[j] += xi[j];
            }
        }

        double f1dim(x)
        double x;
        {
            int j;                                                   250
```

```
        double xt[ML], func();

        for (j=0;j<df;j++) xt[j]=pcom[j]+x*xicom[j];
        return (func(xt));
}

double brent(ax,bx,cx,xmin)
double ax,bx,cx,*xmin;
{
        int iter;                                                    260
        double a,b,d=0.0,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
        double e=0.0;
        double f1dim();

        a=((ax < cx) ? ax : cx);
        b=((ax > cx) ? ax : cx);
        x=w=v=bx;
        fw=fv=fx=f1dim(x);
        for (iter=1;iter<=MAXITER;iter++) {
                printf("\t\tBrent iteration #%d\n", iter);       270
                xm=0.5*(a+b);
                tol2=2.0*(tol1=TOL*fabs(x));
                if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
                        *xmin=x;
                        return fx;
                }
                if (fabs(e) > tol1) {
                        r=(x-w)*(fx-fv);
                        q=(x-v)*(fx-fw);
                        p=(x-v)*q-(x-w)*r;                       280
                        q=2.0*(q-r);
                        if (q > 0.0) p = -p;
                        q=fabs(q);
                        etemp=e;
                        e=d;
                        if (fabs(p) >= fabs(0.5*q*etemp) ||
```

```
                        p <= q*(a-x) || p >= q*(b-x))
                    d=CGOLD*(e=(x >= xm ? a-x : b-x));
                else {
                    d=p/q;                                          290
                    u=x+d;
                    if (u-a < tol2 || b-u < tol2)
                        d=SIGN(tol1,xm-x);
                }
            } else {
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
            }
            u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
            fu=f1dim(u);
            if (fu <= fx) {                                         300
                if (u >= x) a=x; else b=x;
                SHFT(v,w,x,u)
                SHFT(fv,fw,fx,fu)
            } else {
                if (u < x) a=u; else b=u;
                if (fu <= fw || w == x) {
                    v=w;
                    w=u;
                    fv=fw;
                    fw=fu;                                          310
                } else if (fu <= fv || v == x || v == w) {
                    v=u;
                    fv=fu;
                }
            }
        }
        puts("Too many iterations in BRENT");
        exit(0);
        return 0.0;
}                                                                   320

void mnbrak(ax,bx,cx,fa,fb,fc)
```

```
double *ax,*bx,*cx,*fa,*fb,*fc;
{
        double ulim,u,r,q,fu,dum,f1dim();

        *fa=f1dim(*ax);
        *fb=f1dim(*bx);
        if (*fb > *fa) {
                SHFT(dum,*ax,*bx,dum)                                   330
                SHFT(dum,*fb,*fa,dum)
                }
        *cx=(*bx)+GOLD*(*bx-*ax);
        *fc=f1dim(*cx);
        while (*fb > *fc) {
                r=(*bx-*ax)*(*fb-*fc);
                q=(*bx-*cx)*(*fb-*fa);
                u=(*bx)-((*bx-*cx)*q-(*bx-*ax)*r)/
                        (2.0*SIGN(MAX(fabs(q-r),TINY),q-r));
                ulim=(*bx)+GLIMIT*(*cx-*bx);                            340
                if ((*bx-u)*(u-*cx) > 0.0) {
                        fu=f1dim(u);
                        if (fu < *fc) {
                                *ax=(*bx);
                                *bx=u;
                                *fa=(*fb);
                                *fb=fu;
                                return;
                        } else if (fu > *fb) {
                                *cx=u;                                  350
                                *fc=fu;
                                return;
                        }
                        u=(*cx)+GOLD*(*cx-*bx);
                        fu=f1dim(u);
                } else if ((*cx-u)*(u-ulim) > 0.0) {
                        fu=f1dim(u);
                        if (fu < *fc) {
```

```
                        SHFT(*bx,*cx,u,*cx+GOLD*(*cx-*bx))
                        SHFT(*fb,*fc,fu,f1dim(u))                    360
                }
        } else if ((u-ulim)*(ulim-*cx) >= 0.0) {
                u=ulim;
                fu=f1dim(u);
        } else {
                u=(*cx)+GOLD*(*cx-*bx);
                fu=f1dim(u);
        }
        SHFT(*ax,*bx,*cx,u)
        SHFT(*fa,*fb,*fc,fu)                                         370
    }
}

double func(p)
        double p[];
{
        int i;
        double sum1, sum2, sum;
        complex r[ML], to_g_anal();
                                                                     380
        for (i=0; i<df/2; i++) (l[u[i]]).bulk = cmplx(p[2*i], p[2*i+1]);
        for (i=0; i<sen; i++) r[i] = gp(n, l, s+i);

        for(sum1=0.0, sum2=0.0, i=0; i<sen; i++) {
                sum1 += SQR(ccabs(minus(to_g_anal(r[i]), to_g_anal(meas[i]))));
                sum2 += SQR(ccabs(plus(to_g_anal(r[i]), to_g_anal(meas[i]))));
                }
        sum = sqrt(sum1/sum2);

        printf("func({");                                            390
        for (i=0; i<df-1; i++) printf("%g, ", p[i]);
        printf("%g}) = %g\n", p[df-1], sum);

        return (sum);
```

```
}

complex to_g_anal(z)
        complex z;
{
        double r, theta;                                                400

        r = pow(10.0, re(z)/20.0);
        theta = im(z)*PI/180.0;

        return(cmplx(r*cos(theta), r*sin(theta)));
}
```

```
/* This is a version of the multivariable parameter estimation program,
   est.c, which allows one unknown layer, but the layer is
   assumed to be INHOMOGENEOUS. A certain profile function m(x) is
   specified and it is assumed that the permittivity and the conductivity
   of this layer follow the prescribed profile function in the
   following manner:

   e* = e' - je" = e - js/w
   e*(x) = e_inf + A((1/w)*m(x))^-gamma

   where e is permittivity, s is conductivity, w is frequency, e_inf is
   the normalized permittivity for w->inf'ty, gamma is the dispersive
   slope (-1 < gamma < 0), and m(x) is the profile function, which is:

   m(x) = 1 + 4/PI*atan(B)*[sol(D0*10^D, x) - 0.5]

   where sol is the solution to the diffusion equation. There are three
   unknown quantities: A, B, D; (A is complex).

   Yanko Sheiretov          2/7/94          3/30/94                 */

/* For a variety of mathematical reasons, instead of searching for the
   three unknown parameters A, B, and D, we now search for the complex
   epsilon at the two ends of the sample and at a point inside. The rest
   of the sample's epsilon is interpolated with the function given above.
   The diffusion constant D is estimated from the initial slope of the
   profile function.

   Yanko Sheiretov          4/1/94                                  */

#include "complex.h"
#include "objects.h"
#include "matrix.h"
#include <stdio.h>
```

```
#include <math.h>

void chipinfo(), layinfo(), chipinfoout(), layinfoout(), trans();

#define DF      3               /* degrees of freedom */
#define MLSUB 50                /* maximum number of sublayers */
#define MF      200             /* maximum number of Fourier modes */
#define PI      3.141592654
#define E0      8.854e-12       /* permittivity of vacuum */
#define LIM     1.0e-6          /* limit for the convergence test */
#define MAXK 25                 /* number of collocation points */
#define MAXN 100                /* number of Fourier terms */
#define MAXITER 25              /* maximum number of iterations */
#define D0      1.0e-8          /* normalizing factor for D */
#define TOLR  0.01              /* result tolerance */
#define TOLP  0.01              /* parameter tolerance */
#define MAXDAMP 5               /* maximum number of damping steps */
#define NEWJAC 5                /* how often a new jacobian is calculated */
#define INCM  0.01              /* multiplicative increment for derivatives */
#define INCA  0.1               /* additive increment for derivatives */

main()
{
        double  w1;     /* AC frequency */
        complex ea;     /* complex bulk permittivity above electrodes */
        complex sea;    /* complex surface permittivity above electrodes */
                                /* The above two quantities are assumes to be
                                the same for all sensors */
        char    name[20];       /* output file name */
        FILE    *fp;            /* output file pointer */
        int     n;              /* number of layers */
        int     nsub;           /* number of sublayers */
        int     u;              /* index number of unknown sublayer */
        int     iter;           /* number of iterations */
        complex jac[DF][DF];    /* jacobian */
        complex ijac[DF][DF];   /* inverse jacobian */
```

```
        complex old[DF];            /* old guess vector */
        complex new[DF];            /* new guess vector */
        complex res[DF];            /* result vector */
        complex meas[DF];           /* measured values vector */
        struct sensor s[DF];        /* array of sensors */
        struct layer l[ML+MLSUB]; /* array of layers */
        int     i, m;               /* counters */
        complex r;                  /* result register */
        double sum, newsum, oldsum=0.0, sum1, sum2; /* sums */
        double d;                   /* unknown layer thickness */                    80
        double gamma;               /* dispersive log-log slope */
        double einf;                /* epsilon infinity */
        double grsub[MLSUB+1]; /* sublayer grid points */
        void    grsubinit();        /* initializes the sublayer grid */
        double sol();               /* an infinite sum of decaying sinusoids */
        char    newjac=1;           /* whether a new jacobian is needed */
        int     damp=0;             /* damping counter */
        int     last = -1;          /* iteration number for the last time a new
                                           jacobian was calculated */
        complex dx, to_g_anal();                                                     90

/************** Get information from user ****************************/

        fputs("Please enter output file name:  ", stdout);
        scanf("%s", name);
        if((fp = fopen(name, "w")) == NULL) {
                fprintf(stderr, "est error:  can't open %s\n", name);
                exit(); }
        fputs("Please enter the AC frequency [Hz]:  ",stdout);
        scanf("%lf", &w1);                                                           100
        w1 *= 2.0*PI;    /* convert to rad/s */
        fputs("Please enter the bulk permittivity above the ", stdout);
        puts("electrodes [F/m],[S/m]:");
        scanf("%lf,%lf", &re(ea), &im(ea));
        fputs("Please enter the surface permittivity above the ", stdout);
        puts("electrodes [F],[S]:");
```

```
scanf("%lf,%lf", &re(sea), &im(sea));
fputs("\nPlease enter the number of layers:  ", stdout);
scanf("%d", &n);
fputs("Please enter the index number of the unknown layer:  ", stdout);
scanf("%d", &u);
fputs("Please enter the number of sublayers, into which the ", stdout);
fputs("unknown layer\nis to be divided:  ", stdout);
scanf("%d", &nsub);
for(i=0; i<DF; i++) {
        printf("Please enter information about sensor number %d:\n",i);
        chipinfo(s+i); }
puts("\nNow enter information about the layers.");
fputs("Layer number 0 is the topmost layer, ", stdout);
puts("i.e the infinite half space.");
for(i=0; i<n+nsub-1; i++) {
        printf("Please enter information about layer number %d:\n",
                        i <= u ? i : i - nsub + 1);
        if (i != u) layinfo(l+i);
        else {
                i += nsub-1;
                puts("This is the unknown layer.");
                fputs("\tdispersive log-log slope:  ", stdout);
                scanf("%lf", &gamma);
                fputs("\tepsilon infinity:  ", stdout);
                scanf("%lf", &einf);
                fputs("\tlayer thickness [m]:  ", stdout);
                scanf("%lf", &d);
                fputs("\tsurface permittivity and cond", stdout);
                fputs("uctivity [F], [S]: ", stdout);
                scanf("%lf,%lf", &re((l+i)->surface),
                                        &im((l+i)->surface));
                fputs("Please input initial guesses for ", stdout);
                puts("the normalized parameters:");
                fputs("\tA = ", stdout);
                scanf("%lf,%lf", &re(new[0]), &im(new[0]));
                fputs("\tB = ", stdout);
```

```
                scanf("%lf", &re(new[1])); im(new[1]) = 0.0;
                printf("\tD = %gx10^", D0);
                scanf("%lf", &re(new[2])); im(new[2]) = 0.0;
                }
        }
    puts("\nPlease enter the measured gain and phase for :  ([dB],[deg])");
    for(i=0; i<DF; i++) {
        printf("\tsensor number %d :  ", i);                                  150
        scanf("%lf,%lf", &re(meas[i]), &im(meas[i])); }
    puts("\nThe process now begins ...");

/****************** Print out input data ********************************/

    fprintf(fp, "Output file name:  %s\n", name);
    fprintf(fp, "Number of collocation points:  %d\n", MAXK);
    fprintf(fp, "Number of Fourier terms:  %d\n", MAXN);
    fprintf(fp, "Maximum number of iterations:  %d\n", MAXITER);
    fprintf(fp, "AC frequency [Hz]:  %g\n", w1/PI/2.0);                       160
    fprintf(fp,
        "Bulk permittivity above the electrodes [F/m],[S/m]:  %g,%g\n",
            re(ea), im(ea));
    fprintf(fp,
        "Surface permittivity above the electrodes [F],[S]: %g,%g\n",
            re(sea), im(sea));
    fprintf(fp, "Number of layers:  %d\n", n);
    fprintf(fp, "Number of sublayers:  %d\n", nsub);
    fprintf(fp, "Index numbers of the unknown layer:  %d\n", u);
    for(i=0; i<DF; i++) {                                                     170
        fprintf(fp, "Information about sensor number %d:\n",i);
        chipinfoout(s+i, fp); }
    for(i=0; i<n+nsub-1; i++) {
        fprintf(fp, "Information about layer number %d:\n",
                    i<=u ? i : i - nsub + 1);
        if (i != u) layinfoout(l+i, fp);
        else {
            i += nsub-1;
```

```
                fprintf(fp, "This is the unknown layer.\n");
                fprintf(fp, "\tdispersive log-log slope:  %g\n", gamma);
                fprintf(fp, "\tepsilon infinity:  %g\n", einf);
                fprintf(fp, "\tlayer thickness [m]:  %g\n", d);
        fprintf(fp,
          "\tsurface permittivity and conductivity [F],[S]: %g,%g\n",
                        re((l+i)->surface), im((l+i)->surface));
        fputs("\tinitial guesses:\n", fp);
        fprintf(fp, "\t\tA = (%g,%g)\n", re(new[0]), im(new[0]));
        fprintf(fp, "\t\tB = (%g,%g)\n", re(new[1]), im(new[1]));
        fprintf(fp, "\t\tD = %gx10^(%g,%g)\n", D0, re(new[2]),
                                        im(new[2]));
        }
}

fputs("Measured gain and phase for :  ([dB],[deg])\n", fp);
for(i=0; i<DF; i++)
        fprintf(fp, "\tsensor number %d :  %g,%g\n", i,
                        re(meas[i]), im(meas[i]));
fflush(fp);

/************  Fill in remaining sensor and layer data  ***************/

for(i=0; i<DF; i++) {
        (s[i]).k = MAXK;
        (s[i]).N = MAXN;
        (s[i]).en0 = 0.0;           /* No superimposed field assumed */
        (s[i]).w1 = w1;
        (s[i]).ea = ea;
        (s[i]).sea = sea; }

grsubinit(grsub, nsub, d);          /* Initialize sublayer grid */

/* fill in sublayer thicknesses and surface parameters */
for (i=0; i<nsub-1; i++)            {
        (l+i+u)->thickness = (grsub[i+1] - grsub[i]);
```

```
            re((l+u+i)->surface) = im((l+u+i)->surface) = 0.0;
            }       /* within sublayers there are no surface par. */
        (l+u+nsub-1)->thickness = (grsub[nsub] - grsub[nsub-1]);

        /* convert measured data to analytical form */
        for (i=0; i<DF; i++) meas[i] = to_g_anal(meas[i]);                    220

/******************  Iteration process  ******************************/

        /* iteration control loop */
        for(iter=1; iter <= MAXITER; iter++) {
                printf("Iteration number %d\n", iter);
                for (i=0; i<DF; i++)
                        printf("new[%d] = (%g,%g)\n",i,re(new[i]), im(new[i]));

                if (newjac && last == iter-1) {                               230
                        printf("Convergence failure:  Badness %g > %g\n",
                                oldsum, TOLR);
                        fprintf(fp, "Convergence failure:  Badness %g > %g\n",
                                oldsum, TOLR);
                        exit(0);
                        }

                /* calculate res */
                trans(new, nsub, grsub, l, u, d, w1, einf, gamma);
                for(m=0; m<DF; m++) {                                         240
                   res[m] = to_g_anal(gp(n+nsub-1, l, s+m));
                   printf("res[%d] = (%g,%g)\n", m, re(res[m]), im(res[m]));
                   }

                /* test to see whether res is close to meas */
                for(sum1=0.0,sum2=0.0,i=0; i<DF; i++) {
                        newsum = ccabs(minus(res[i], meas[i]));
                        sum1 += sq(newsum);
                        newsum = ccabs(plus(res[i], meas[i]));
                        sum2 += sq(newsum);                                   250
```

```
        }
newsum = sqrt(sum1/sum2);

if(newsum < TOLR) {
        printf("res is close enough to meas:  \n");
        printf("\tnewsum = %g < %g\n", newsum, TOLR);
        break; }

/* test to see whether damping is needed */
if (newsum > oldsum && !newjac) {                                    260
        iter--;          /* does not count as an iteration */
        printf ("Damping needed:  step #%d\n", damp+1);
        printf ("because %g > %g\n", newsum, oldsum);
        if (damp >= MAXDAMP) { /* new jacobian is needed */
                printf("Even worse -- new jacobian!\n");
                newjac = 1;
                for (i=0; i<DF; i++) new[i] = old[i];
                continue;
                }
        else {                                                       270
                damp++;
                for (i=0; i<DF; i++)
                   new[i] = scale(plus(new[i], old[i]), 0.5);
                continue;
                }
        }
else {
        oldsum = newsum;
        damp = 0;
        }                                                            280

/* calculate jacobian partial derivatives */
if (iter-last >= NEWJAC) newjac = 1;
if (newjac) puts("\nJacobian calculation !");
if (newjac) for(m=0; m<DF; m++) {
        if (ccabs(new[m]) != 0.0) dx = scale(new[m], INCM);
```

```
        else dx = cmplx(INCA, 0.0);
        new[m] = plus(new[m], dx);
        printf("Jac:  new[%d] is now (%g,%g)\n", m,
                        re(new[m]), im(new[m]));                    290
        /* place new values in layers */
        trans(new, nsub, grsub, l, u, d, w1, einf, gamma);
        /* evaluate the derivatives */
        for(i=0; i<DF; i++) {
                r = to_g_anal(gp(n+nsub-1, l, s+i));
                jac[i][m] = over(minus(r, res[i]), dx);
                printf("r[%d] = (%g,%g)\n", i, re(r), im(r));
                printf("jac[%d][%d] = (%g,%g)\n",
                        i, m, re(jac[i][m]), im(jac[i][m]));
                }                                                   300
        new[m] = minus(new[m], dx);        /* restore the value */
        }

/* new becomes old */
/* subtract meas from res */
for(i=0; i<DF; i++) {
        old[i] = new[i];
        res[i] = minus(res[i], meas[i]);
        }
                                                                    310
/* calculate correction vector */
if (newjac) invert(jac, ijac, DF);
mulv(ijac, res, new, DF);
if (newjac) last = iter;
newjac = 0;

/* find new values for new */
for(i=0; i<DF; i++) new[i] = minus(old[i], new[i]);

/* test to see if new is close to old */                           320
for(i=0, sum=0.0; i<DF; i++)
        sum += ccabs(minus(new[i], old[i]));
```

```
                if(sum < TOLP) {
                        printf("new is close enough to old:  \n");
                        printf("\tsum = %g < %g\n", sum, TOLP);
                        break; }

                /* End of iteration cycle */
                fflush(stdout);
                }                                                          330

/****************** Print Out Results ********************************/

        if (iter > MAXITER) {
                puts("Problems with convergence.");
                fputs("Problems with convergence.\n", fp);
                }
        else {
                fputs("\nResults\n", fp);
                fprintf(fp, "\t\tA = (%g,%g)\n", re(new[0]), im(new[0]));    340
                fprintf(fp, "\t\tB = (%g,%g)\n", re(new[1]), im(new[1]));
                fprintf(fp, "\t\tD = %gx10^(%g,%g)\n", D0, re(new[2]),
                                                        im(new[2]));

                puts("\nResults:");
                printf("\t\tA = (%g,%g)\n", re(new[0]), im(new[0]));
                printf("\t\tB = (%g,%g)\n", re(new[1]), im(new[1]));
                printf("\t\tD = %gx10^(%g,%g)\n", D0, re(new[2]),im(new[2]));
                }
                                                                           350
        fclose(fp);
        fflush(stdout);
}


/* The following function is a solution to the diffusion equation:

  dn     ddn
```

```
   -- = D ----   with the boundary conditions  n(x=0, t) = n_0
   dt    dxdx                                                              360

                                        dn
                                  and   -- (x=d, t) = 0
                                        dx

   The solutions are decaying sines.                                    */

double sol (x, d, D)
        double x;       /* x - position */
        double d;       /* layer width */
        double D;       /* normalized diffusion constant */               370
{
        double sum = 0.0, inc;
        int m;

        d *= 2.0;
        for(m=1; m<2*MF; m+=2) {
                inc = PI*(double)m;
                inc = 4.0*exp(-D*inc*inc/d/d)*sin(inc*x/d)/inc;
                if (fabs(inc) < LIM) break;         /* if a limit is approached */
                sum += inc;                                                 380
                }
        if(inc >= LIM) {
                printf("sol() warning!  - %d Fourier terms were ", 2*MF);
                printf("calculated and still %g > %g\n", inc, LIM);
                }
        return sum;
}


/* The following function initializes the sublayer grid */                 390

void grsubinit(grsub, nsub, d)
        double *grsub;  /* array of grid points */
        int    nsub;    /* number of sublayers */
```

```
        double d;       /* layer thickness */
{
        int i;

        for(i=0; i<=nsub; i++) grsub[i] = d*sin(PI/2.0*(double)i/(double)nsub);
        /* a sinusoidal distribution is assumed, making the points closer to
           the sensor more densely spaced. See gridinit() in coef.c  */
}

/* place values for the parameters of all sublayers */
void trans(new, nsub, grsub, l, u, d, w1, einf, gamma)
        complex *new;
        int nsub, u;
        double grsub[], d, w1, einf, gamma;
        struct layer *l;
{
        int i;
        double x, temp;
        complex to_e_stan();

        for (i=0; i<nsub; i++) {
                x = grsub[i] + 0.5*((l+u+i)->thickness);
                temp = sol(x, d, D0*pow(10.0, re(new[2]))) - 0.5;
                temp = 1.0 + temp*4.0/PI*atan(re(new[1]));
                temp = pow(temp/w1, -gamma);
                (l+u+i)->bulk = to_e_stan(scale(plus(cmplx(einf, 0.0),
                        scale(new[0], temp)), E0), w1);
                }
}

complex to_g_anal(z)
        complex z;
{
        double r, theta;

        r = pow(10.0, re(z)/20.0);
```

```
        theta = im(z)*PI/180.0;

        return(cmplx(r*cos(theta), r*sin(theta)));
}

complex to_e_stan(z, w)
        complex z;
        double w;
{
        return (cmplx(re(z), -w*im(z)));                                    440
}
```

```
/* This version of the parameter estimation program "est.c" uses a different
   approach: instead of searching for a root, we search for a minimum of an
   error function (func()). This has the advantage that the numerical method
   is more stable. Also, it allows for having more sensors than unknown layers
   and does not fail if due to experimental errors no roots exist. Preliminary
   tests seem to indicate that it takes more computation time than "est.c".

   It is different from "estm.c" in that it uses the amoeba method.

   Yanko Sheiretov        3/28/94                                    */

#include "complex.h"
#include "objects.h"
#include <stdio.h>
#include <math.h>

#define FTOL   0.001    /* Fractional tolerance for the error function */
#define ATOL   0.01     /* Absolute tolerance for the error function */
#define UV     2e-12    /* Unit vector for creating initial simplex */
#define SQR(a) (sqrarg=(a),sqrarg*sqrarg)
#define PI     3.141592654
#define MAXK   25       /* number of collocation points */
#define MAXN   100      /* number of Fourier terms */
#define MAXITER 500     /* maximum number of iterations */
#define ALPHA 1.0
#define BETA 0.5
#define GAMMA 2.0

int      n;             /* number of layers */
int      u[ML];         /* index numbers for unknown layers */
complex  meas[ML];      /* measured values vector */
struct   sensor s[ML];  /* array of sensors */
struct   layer l[ML];   /* array of layers */
int df,sen;
```

```
double sqrarg;
void chipinfo(), layinfo(), chipinfoout(), layinfoout();

main()
{
        int i, j;           /* counters */                                  40
        double min,func();
        double  w1;         /* AC frequency */
        complex ea;         /* complex bulk permittivity above electrodes */
        complex sea;        /* complex surface permittivity above electrodes */
                                /* The above two quantities are assumes to be
                                        the same for all sensors */
        char    name[20];       /* output file name */
        FILE    *fpout;         /* output file pointer */
        int     ilo, amoeba();
        double  p[ML+1][ML], y[ML+1];                                       50

/*************** Get information from user ****************************/

        fputs("Please enter output file name:  ", stdout);
        scanf("%s", name);
        if((fpout = fopen(name, "w")) == NULL) {
                fprintf(stderr, "est error:  can't open %s\n", name);
                exit(); }
        fputs("Please enter the AC frequency [Hz]:  ", stdout);
        scanf("%lf", &w1);                                                  60
        w1 *= 2.0*PI;     /* convert to rad/s */
        fputs("Please enter the bulk permittivity above the ", stdout);
        puts("electrodes [F/m],[S/m]:");
        scanf("%lf,%lf", &re(ea), &im(ea));
        fputs("Please enter the surface permittivity above the ", stdout);
        puts("electrodes [F],[S]:");
        scanf("%lf,%lf", &re(sea), &im(sea));
        fputs("\nPlease enter the number of layers:  ", stdout);
        scanf("%d", &n);
        fputs("Please enter the number of sensors:  ", stdout);             70
```

```
        scanf("%d", &sen);
        puts("Please enter the number of unknown layers.  Should be less than");
        fputs("or equal to the number of sensors:  ", stdout);
        scanf("%d", &df);
        df *= 2;
        puts("Please enter the index numbers of the unknown layers,");
        puts("separated with commas, starting from the infinite half");
        puts("space (number 0), e.g.  2,3,5");
        for(i=0; i<df/2; i++) {
                scanf("%d", u+i);                                          80
                if (i < df/2-1) while (getchar() != ',',); }
        putchar('\n');

        for(i=0; i<sen; i++) {
                printf("Please enter information about sensor number %d:\n",i);
                chipinfo(s+i); }
        puts("\nNow enter information about the layers.  The values entered");
        puts("for the bulk properties of the unknown layers will be used as");
        puts("the initial guess.  Layer number 0 is the topmost layer,");
        puts("i.e the infinite half space.");                              90
        for(i=0; i<n; i++) {
                printf("Please enter information about layer number %d:\n", i);
                layinfo(l+i); }

        puts("\nPlease enter the measured gain and phase for :  ([dB],[deg])");
        for(i=0; i<sen; i++) {
                printf("\tsensor number %d :  ", i);
                scanf("%lf,%lf", &re(meas[i]), &im(meas[i])); }
        puts("\nThe process now begins ...");

                                                                          100
/******************* Print out input data *********************************/

        fprintf(fpout, "Output file name:  %s\n", name);
        fprintf(fpout, "Number of collocation points:  %d\n", MAXK);
        fprintf(fpout, "Number of Fourier terms:  %d\n", MAXN);
        fprintf(fpout, "Maximum number of iterations:  %d\n", MAXITER);
```

```
        fprintf(fpout, "AC frequency [Hz]:  %g\n", w1/PI/2.0);
        fprintf(fpout,
                "Bulk permittivity above the electrodes [F/m],[S/m]:  %g,%g\n",
                re(ea), im(ea));                                              110
        fprintf(fpout,
                "Surface permittivity above the electrodes [F],[S]: %g,%g\n",
                re(sea), im(sea));
        fprintf(fpout, "Number of layers:  %d\n", n);
        fprintf(fpout, "Number of sensors:  %d\n", sen);
        fprintf(fpout, "number of unknown layers:  %d\n", df/2);
        fputs("Index numbers of the unknown layers:  ", fpout);
        for(i=0;i<df/2;i++) {
                if (i<df/2-1) fprintf(fpout, "%d, ", u[i]);
                else fprintf(fpout, "%d\n", u[i]); }                          120
        for(i=0; i<sen; i++) {
                fprintf(fpout, "Information about sensor number %d:\n", i);
                chipinfoout(s+i, fpout); }
        for(i=0; i<n; i++) {
                fprintf(fpout, "Information about layer number %d:\n", i);
                layinfoout(l+i, fpout); }

        fputs("Measured gain and phase for :  ([dB],[deg])\n", fpout);
        for(i=0; i<sen; i++) fprintf(fpout, "\tsensor number %d :  %g,%g\n", i,
                                        re(meas[i]), im(meas[i]));            130
        fflush(fpout);

/********************  Fill in remaining sensor data **********************/

        for(i=0; i<sen; i++) {
                (s[i]).k = MAXK;
                (s[i]).N = MAXN;
                (s[i]).en0 = 0.0;           /* No superimposed field assumed */
                (s[i]).w1 = w1;
                (s[i]).ea = ea;                                               140
                (s[i]).sea = sea; }
```

```
/****************** Initialize p and y ***********************************/

        for(i=0; i<df; i++)
                p[0][i] = i%2==0 ? re((l[u[i/2]]).bulk) : im((l[u[i/2]]).bulk);
        for (i=0; i<df; i++)
                for (j=0; j<df; j++) p[i+1][j] = i!=j ? p[0][j] : p[0][j]+UV;
        for (i=0; i<df+1; i++) y[i] = func(p[i]);

/****************** DO THE THING *************************************/

        ilo = amoeba(p, y);                     /* That's IT, folks! */
        min = y[ilo];

/****************** Print Out Results ************************************/

        /* place results into layer structures */
        for(i=0; i<df/2; i++)
                (l[u[i]]).bulk = cmplx(p[ilo][2*i], p[ilo][2*i+1]);

        puts("Done -- see output file for results.");
        printf("Minimum achieved:  %g\n\n", min);
        fprintf(fpout, "Minimum achieved:  %g\n\n", min);
        fprintf(fpout, "\n%15s%15s%15s\n", "Layer number",
                        "Permittivity", "Conductivity");
        for(i=0; i<46; i++) putc('-', fpout);
        fputs("\n$\n", fpout);
        for (i=0; i<df/2; i++)
                fprintf(fpout, "%15d%15e%15e\n", u[i], re((l[u[i]]).bulk),
                                        im((l[u[i]]).bulk));
        fputs("$\n", fpout);

        fclose(fpout);
        fflush(stdout);
}

double func(p)
```

```
        double p[];
{                                                                    180
        int i;
        double sum1, sum2, sum;
        complex r[ML], to_g_anal();

        for (i=0; i<df/2; i++) (l[u[i]]).bulk = cmplx(p[2*i], p[2*i+1]);
        for (i=0; i<sen; i++) r[i] = gp(n, l, s+i);

        for(sum1=0.0, sum2=0.0, i=0; i<sen; i++) {
                sum1 += SQR(ccabs(minus(to_g_anal(r[i]), to_g_anal(meas[i]))));
                sum2 += SQR(ccabs(plus(to_g_anal(r[i]), to_g_anal(meas[i]))));   190
                }
        sum = sqrt(sum1/sum2);

        printf("func({");
        for (i=0; i<df-1; i++) printf("%g, ", p[i]);
        printf("%g}) = %g\n", p[df-1], sum);

        return (sum);
}
                                                                     200
complex to_g_anal(z)
        complex z;
{
        double r, theta;

        r = pow(10.0, re(z)/20.0);
        theta = im(z)*PI/180.0;

        return(cmplx(r*cos(theta), r*sin(theta)));
}                                                                    210

#define GET_PSUM for (j=0;j<df;j++) { for (i=0,sum=0.0;i<mpts;i++)\
                                                sum += p[i][j]; psum[j]=sum;}
int amoeba(p,y)
```

```
double p[][ML],y[];
{
        int i,j,ilo,ihi,inhi,nfunk,mpts=df+1;
        double ytry,ysave,sum,rtol,amotry(),psum[ML];

        nfunk=0;                                                    220
        GET_PSUM
        for (;;) {
                ilo=0;
                ihi = y[0]>y[1] ? (inhi=1,0) : (inhi=0,1);
                for (i=0;i<mpts;i++) {
                        if (y[i] < y[ilo]) ilo=i;
                        if (y[i] > y[ihi]) {
                                inhi=ihi;
                                ihi=i;
                        } else if (y[i] > y[inhi])                  230
                                if (i != ihi) inhi=i;
                }
                rtol=2.0*fabs(y[ihi]-y[ilo])/(fabs(y[ihi])+fabs(y[ilo]));
                if (rtol < FTOL || 0.5*(y[ihi]+y[ilo]) < ATOL) return ilo;
                if (nfunk >= MAXITER) {
                        puts("Too many iterations in AMOEBA");
                        exit(0);
                        }
                ytry=amotry(p,y,psum,ihi,&nfunk,-ALPHA);
                if (ytry <= y[ilo])                                 240
                        ytry=amotry(p,y,psum,ihi,&nfunk,GAMMA);
                else if (ytry >= y[inhi]) {
                        ysave=y[ihi];
                        ytry=amotry(p,y,psum,ihi,&nfunk,BETA);
                        if (ytry >= ysave) {
                           for (i=0;i<mpts;i++) {
                                if (i != ilo) {
                                   for (j=0;j<df;j++) {
                                        psum[j]=0.5*(p[i][j]+p[ilo][j]);
                                        p[i][j]=psum[j];            250
```

```
                                        }
                                    y[i]=func(psum);
                                    }
                                }
                            nfunk += df;
                            printf("Iteration counter:  %d\n", nfunk);
                            GET_PSUM
                        }
                    }
                }                                                          260
}

double amotry(p,y,psum,ihi,nfunk,fac)
double p[][ML],y[],psum[],fac;
int ihi,*nfunk;
{
        int j;
        double fac1,fac2,ytry,ptry[ML];

        fac1=(1.0-fac)/df;                                                 270
        fac2=fac1-fac;
        for (j=0;j<df;j++) ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
        ytry=func(ptry);
        ++(*nfunk);
        printf("Iteration counter:  %d\n", *nfunk);
        if (ytry < y[ihi]) {
                y[ihi]=ytry;
                for (j=0;j<df;j++) {
                        psum[j] += ptry[j]-p[ihi][j];
                        p[ihi][j]=ptry[j];                                 280
                }
        }
        return ytry;
}
```

getgp.c

```
/* This program dvives gp by providing user input. Analogous to mz1x
   Yanko Sheiretov          1/24/94                                    */

#include "complex.h"
#include "objects.h"
#define PI      3.141592654

main()
{
        sensor s;                                                      10
        layer l[MAXLAYERS];
        complex result;
        void    chipinfo(), layinfo();
        int n, i;

        fputs("Please enter the number of collocation points:  ", stdout);
        scanf("%d", &s.k);
        fputs("Please enter the number of Fourier terms:  ", stdout);
        scanf("%d", &s.N);
        fputs("Please enter the AC frequency [Hz]:  ", stdout);        20
        scanf("%lf", &s.w1);
        s.w1 *= 2.0*PI;  /* convert to rad/s */
        fputs("Please enter the bulk permittivity above the ", stdout);
        puts("electrodes [F/m],[S/m]:");
        scanf("%lf,%lf", &re(s.ea), &im(s.ea));
        fputs("Please enter the surface permittivity above the ", stdout);
        puts("electrodes [F],[S]:");
        scanf("%lf,%lf", &re(s.sea), &im(s.sea));
        s.en0 = 0.0;
        fputs("\nPlease enter the number of layers:  ", stdout);       30
        scanf("%d", &n);
        puts("Please enter information about the sensor:");
        chipinfo(&s);
        fputs("\nNow enter information about the layers.  ", stdout);
```

```
        puts("Layer number 1 is the topmost layer.");
        for(i=1; i<=n; i++) {
                printf("Please enter information about layer number %d:\n", i);
                layinfo(l+i); }

        result = gp(n, l, &s);                                                    40
        printf("gain = %g \tphase = %g\n", re(result), im(result));
}
```

H.5 Subsidiary Parameter Estimation Routines

admit.c

```
/* This function calculates y11 and y12.
   Yanko Sheiretov        1/19/94        1/25/94        */

#include "est.h"

void    admit(v, c)
        complex v[N1];
        complex c[MF];
{
        complex temp, temp1, temp2;                                    10
        int     m, j;
        double  ff, da, db, dc;

        temp = y12 = y11 = cmplx(0.0, 0.0);
        temp1 = temp2 = cmplx(1.0, 0.0);
        for (m=1;m<=N;m++) {
                temp1 = cmplx(0.0, 0.0);
                ff = 1.0/sq(PI)/m;
                for (j=0; j<k; j++) {
                        da = 1.0/(grid(j+2) - grid(j+1));              20
                        db = 1.0/(grid(j+1) - grid(j));
                        dc = (da+db)*cos(2.0*PI*m*grid(j+1));
                        dc -= da*cos(2.0*PI*m*grid(j+2));
                        dc -= db*cos(2.0*PI*m*grid(j));
                        dc *= ff/m;
                        temp1 = plus(temp1, scale(v[j], dc));
                        }
                dc = ff*(cos(2.0*PI*m*grid(0))-cos(2.0*PI*m*grid(1)))/
                                ((grid(1) - grid(0))*m);
                re(temp1) += dc;                                       30
                temp1 = times(temp1, c[m]);
                dc = sin(2.0*PI*m*grid(k+1));
```

```
            temp2 = scale(temp1, dc);
            dc = sin(2.0*PI*m*grid(0)) - sin(2.0*PI*m*grid(k+1));
            y11 = plus(y11, scale(temp1, dc));
            y12 = plus(y12, temp2);
            }
        temp1 = cmplx(0.0, 0.0);
        for(j=0; j<k; j++) temp1 = plus(temp1, scale(v[j],
                                        grid(j+2)-grid(j)));                    40
        re(temp1) += grid(0) + grid(1);
        temp1 = scale(temp1, 1.0/h);
        if (en0 != 0.0) temp1 = plus(temp1, scale(rea, en0));
        y11 = plus(y11, scale(temp1, grid(0) - grid(k+1) + 0.5));
        temp = scale(temp1, 0.5 - grid(k+1));
        y12 = minus(y12, temp);
        temp = scale(v[k-1], 1.0/(grid(k+1)-grid(k)));
        temp1 = scale(minus(cmplx(1.0, 0.0), v[0]), 1.0/(grid(1) - grid(0)));
        temp1 = times(minus(temp1, temp), rsea);
        y11 = scale(plus(y11, temp1), 2.0);                                     50
        temp = times(temp, rsea);
        y12 = scale(plus(y12, temp), 2.0);
/*      printf("y11 = (%g,%g)\n", re(y11), im(y11));        */
/*      printf("y12 = (%g,%g)\n", re(y12), im(y12));        */
}
```

coef.c

```
/* This function computes the matrix elements
   Yanko Sheiretov        1/19/94          1/25/94                  */

#include "est.h"

void coef(a, x, c)
        complex a[N1][N1];              /* matrix coefficients */
        complex x[N1];                  /* right-hand vector */
        complex c[MF];                  /* total capacitance density */
{                                                                    10
        int     flag=0, r, cc;
        double  dpi, y0, y1, s1, s2, da, db, dc;
        complex temp;

        dpi = 2.0*PI;
        y0 = grid(0);
        y1 = grid(1);

        for(r=0; r<k; r++) {
                s1 = r+1==k ? grid(k+1) : (grid(r+2) + grid(r+1))/2.0;      20
                s2 = r==0 ? y0 : (grid(r+1)+grid(r))/2.0;
                if (k%2 == 0 && r == k/2) flag=1;
                for(cc=0;cc<k;cc++) {
                        if (k%2==1 && r==(k+1)/2-1 && cc==(k+3)/2-1) flag=1;
                        if (flag==1) a[r][cc] = a[k-1-r][k-1-cc];
                        else {
                                da = 1.0/(grid(cc+2) - grid(cc+1));
                                db = 1.0/(grid(cc+1) - grid(cc));
                                temp = fhsum(dpi*(s1 + grid(cc+1)), c);
                                temp=plus(temp, fhsum(dpi*(s1-grid(cc+1)),c));   30
                                temp=minus(temp, fhsum(dpi*(s2+grid(cc+1)),c));
                                temp=minus(temp, fhsum(dpi*(s2-grid(cc+1)),c));
                                a[r][cc] = scale(temp, da+db);
                                temp = fhsum(dpi*(s2 + grid(cc+2)), c);
```

```
                                temp =plus(temp,fhsum(dpi*(s2-grid(cc+2)),c));
                                temp =minus(temp,fhsum(dpi*(s1+grid(cc+2)),c));
                                temp =minus(temp,fhsum(dpi*(s1-grid(cc+2)),c));
                                a[r][cc] = plus(a[r][cc], scale(temp, da));
                                temp = fhsum(dpi*(s2 + grid(cc)), c);
                                temp =plus(temp,fhsum(dpi*(s2-grid(cc)),c));                    40
                                temp =minus(temp,fhsum(dpi*(s1+grid(cc)),c));
                                temp =minus(temp,fhsum(dpi*(s1-grid(cc)),c));
                                a[r][cc] = plus(a[r][cc], scale(temp, db));
                                a[r][cc] = scale(a[r][cc], 0.5/sq(PI));
/*      printf("a[%d][%d] = (%g,%g)\n", r, cc, re(a[r][cc]), im(a[r][cc])); */
                                } }
            temp = fhsum(dpi*(s1+y0), c);
            temp = plus(temp, fhsum(dpi*(s1-y0), c));
            temp = plus(temp, fhsum(dpi*(s2+y1), c));
            temp = plus(temp, fhsum(dpi*(s2-y1), c));                                           50
            temp = minus(temp, fhsum(dpi*(s2+y0), c));
            temp = minus(temp, fhsum(dpi*(s2-y0), c));
            temp = minus(temp, fhsum(dpi*(s1+y1), c));
            temp = minus(temp, fhsum(dpi*(s1-y1), c));
            x[r] = scale(temp, -0.5/sq(PI)/(y1-y0));
/*          printf("x[%d] = (%g,%g)\n", r, re(x[r]), im(x[r])); */
            }

        /* The following section computes a2, a3, x2 and x3 */
        for (r=0; r<k; r++) {                                                                   60
                dc =    r==0 ?  (grid(2)+y1)/2.0 - y0 :
                        r==k-1 ? grid(k+1) - (grid(k) + grid(k-1))/2.0 :
                                (grid(r+2) - grid(r))/2.0;
                re(x[r]) -= dc*(y0+y1)/h;
                if (en0 != 0.0) x[r] = minus(x[r], scale(rea, dc*en0));
                for(cc=0; cc<k; cc++) {
                        da = grid(cc+2) - grid(cc);
                        re(a[r][cc]) += dc*da/h;
                        a[r][cc] = plus(a[r][cc], scale(rsea, f1(r, cc)));
                        }                                                                       70
```

```
                if (r==0) x[r] = plus(x[r], scale(rsea, 1.0/(y1-y0)));
                }
}

double f1(r, cc)
        int r, cc;
{
        double da, db, dc;

        da = 1.0/(grid(r+2) - grid(r+1));                                    80
        db = 1.0/(grid(r+1) - grid(r));
        dc = grid(r+2) - grid(r);
        return (r==cc ? dc*db*da : r-cc==1 ? -1.0*db : cc-r==1 ? -1.0*da :0.0);
}

complex fhsum(fx, c)
        double fx;
        complex c[MF];
{
        complex sum;                                                         90
        int i;

        sum = cmplx(0.0, 0.0);
        for (i=1; i<=nmax; i++)
                sum = plus(sum, scale(minus(c[i], ckmin), sin(i*fx)/sq(i)));
        return (plus(sum, scale(ckmin, fsum(fx))));
}


double fsum(fsx)                                                            100
        double fsx;
{
        int sign, i;
        double z, a, b, c, d;
        static double bern[21] = {0.0, .166667, .033333, .023809, .033333,
                .075757, .253113, 1.166666, 7.092156, 54.971177, 529.124242,
```

```
                6192.123188, 8.658025311e4, 1.425517167e6, 2.729823107e7,
                6.015808739e8, 1.511631577e10, 4.296146431e11, 1.371165521e13,
                4.883323190e14, 1.929657934e16};
                                                                        110
        sign = fsx<0.0 ? -1 : 1;
        z = fabs(fsx);
        if (fsx == 0.0) a = 0.0;
        else {
                b = a = z*(log(z) - 1.0);
                i = 1;
                while (fabs(b/a) > 0.0001) {
                        c = bern[i] * pow(z, (2.0*i + 1.0));
                        d = 2.0*i*(2.0*i + 1.0)*fact(2*i);
                        b = c/d;                                        120
                        a -= b;
                        i++;
                        }
                }
        return (-a*sign);
}


double  fact(x)
        int x;                                                          130
{
        int i;
        double r = 1.0;

        for(i=x;i>1;i--) r *= (double)i;
        return r;
}

void    gridinit()              /* this function initializes the grid */
{                                                                       140
        int r;    /* counter */
```

```
        for(r=0; r<=k+2; r++) gr[r] = 0.25 - 0.5*g*cos(PI*r/(double)(k+1));
}
```

```
/* This is the control function for the forward gain/phase calculation.
   Yanko Sheiretov        1/20/94          1/25/94                    */

#include "est.h"

/* Define global parameters */

double lambda;
double eox;      /* substrate permittivity */
double h;        /* thickness */                                      10
double g;        /* interelectrode thickness */
double yload;    /* load impedance */
complex ckmin;
complex rea, rsea;
int     k, N, nmax, num;
double en0;
double w1;       /* frequency */
complex y11, y12;          /* complex lumped admittances */
complex yp;      /* parasitic admittance */
double ap, bp;                                                        20
double gr[N1+2];           /* grid values */

complex gp(n, l, s)
        int       n;       /* number of layers */
        struct layer      l[];      /* array of layers */
        struct sensor     *s;       /* sensor information */
{
        complex c[MF];             /* surface capacitance density */
        complex v[N1];             /* voltage distribution */
        complex a[N1][N1];         /* matrix coefficients */          30
        complex x[N1];             /* result vector */
        complex temp;
        double gain, phase;
```

```
    /* Transfer sensor information to global variables */
    N       = s->N;
    k       = s->k;
    lambda  = s->lambda;
    eox     = s->eox;
    h       = s->h;                                          40
    g       = s->g;
    yload   = s->yload;
    en0     = s->en0;
    w1      = s->w1;
    ap      = s->ap;
    bp      = s->bp;
    num     = n;
    re(rea)  = re(s->ea)/eox;
    im(rea)  = -im(s->ea)/eox/w1;
    re(rsea)= re(s->sea)/eox/lambda;                         50
    im(rsea)= -im(s->sea)/eox/lambda/w1;

    gridinit();          /* initialize grid */

    /* execute actual algorithm */
    scap(l, c);
    coef(a, x, c);
    solve(a, x, v);
    admit(v, c);
                                                             60
    temp = plus(yp, cmplx(yload, 0.0));
    temp = plus(temp, y12);
    temp = plus(temp, y11);
    temp = over(plus(y12, yp), temp);
    gain = 20.0*log10(ccabs(temp));
    phase = 180.0*(atan2(im(temp), re(temp)))/PI;

    return (cmplx(gain, phase));
}
```

scap.c

```
/* This function computes the surface capacitance density for layers
   of homogeneous media.

   Yanko Sheiretov        1/19/94        1/25/94                */

#include "est.h"
#define A12(j,m)        (scale(ne[(j)],2.0*PI*(m)/sinh(2*PI*(m)*d[(j)])))
#define A22(j,m)        (scale(ne[(j)],2.0*PI*(m)/tanh(2*PI*(m)*d[(j)])))

void    scap(l, c)                                                    10
        struct layer l[];           /* array of layer structures */
        complex c[];                /* surface capacitance density */
{
        complex ne[ML];             /* normalized bulk properties */
        complex nse[ML];            /* normalized surface properties */
        double d[ML];               /* normalized thicknesses */
        int     j, m;               /* counters */
        complex tmp, temp, temp1;

        /* normalize quantities */                                    20
        for (j=0; j<num; j++) {
                ne[j] = cmplx(re((l+j)->bulk)/eox,
                                -im((l+j)->bulk)/eox/w1);
                nse[j] = cmplx(re((l+j)->surface)/eox/lambda,
                                -im((l+j)->surface)/eox/lambda/w1);
                d[j] = (l+j)->thickness/lambda;
                }

        /* Implement algorithm */
        for(m=1; m<=N2; m++) {                                        30
                tmp = cmplx(1.0, 0.0);
                c[m] = A22(0,m);
                if (num>1) for (j=1; j<num; j++)
                        /* check to prevent overflow when evaluating sinh */
```

```
                    if (2*PI*m*d[j] < 44.0) {
                            temp = plus(A22(j,m),
                                    scale(nse[j], sq(2.0*PI*m)));
                            ckmin = plus(c[m], temp);
                            temp = A12(j,m);
                            temp = over(csq(temp), ckmin);                    40
                            temp1 = A22(j,m);
                            c[m] = minus(temp1, temp);
                            }
                    else c[m] = A22(j,m);
/*                  printf("C[%d]=(%g,%g)\n", m, re(c[m]), im(c[m])); */
                /* find out when cn/kn approaches a limit */
                if (m>1) {
                        tmp = minus(scale(c[m-1], 0.5/PI/(m-1)),
                                scale(c[m], 0.5/PI/m));
                        if (fabs(im(tmp)) < 1e-20) im(tmp) = 0.0;             50
                        }
                if (ccabs(tmp) < 1e-5) break;       /* limit approached */
                }
        if (m == N2) {
                puts("scap warning:  \t c[m] did not reach a limit.");
                puts("Recompile with a larger N2.");
                }
        nmax = m-1;
        ckmin = scale(c[nmax], 0.5/PI/nmax);
        /* fill up the rest with Ckmin */                                     60
        for(;m<=N2;m++) c[m] = scale(ckmin, 2.0*PI*m);

        /* Compute the parallel addition of surface capacitance densities
        above and below the electrodes */
        for(m=1;m<=N2;m++) {
                c[m] = scale(c[m], 0.5/PI/m);
                re(c[m]) += 1.0/tanh(2*PI*m*h);
                if (m > 1 && ccabs(minus(c[m-1], c[m])) < 1e-5) break;
/*              printf("L[%d] = (%g,%g)\n", m, re(c[m]), im(c[m])); */
                }                                                             70
```

```
        if (m == N2) {
                puts("scap warning:  \t c[m] did not reach a limit.");
                puts("Recompile with a largor N2.");
                }
        nmax = m-1;
        ckmin = c[nmax];
        for(;m<=N2;m++) c[m] = ckmin;

        /* Compute parasitic admittance (as in bodex.for) */
        re(yp) = ap*re(ne[num-2]) + bp;                                    80
        im(yp) = ap*im(ne[num-2]);
}

#undef A12
#undef A22
```

---

solve.c

```
/* This function solves k equations with k unknowns using gaussian
   elimination. Format av=x
   Yanko Sheiretov          1/19/94          1/25/94                  */

#include "est.h"

void    solve(a,x,v)
        complex a[N1][N1], x[N1], v[N1];
{
        complex temp[N1];                                              10
        complex tempx, tempr;
        int     r, c, i, s;

        for(i=0;i<k-1;i++) {
                /* handles the case with zero leading coefficient */
                s = 0;
                while (ccabs(a[i][i]) == 0.0 && s+i < k-1) {
                        tempx = x[i];
                        for(c=i;c<k;c++) temp[c] = a[i][c];
                        for(r=i;r<k-1;r++) {                           20
                                x[r] = x[r+1];
                                for(c=i;c<k;c++) a[r][c] = a[r+1][c];
                                }
                        x[k-1] = tempx;
                        for (c=i;c<k;c++) a[k-1][c] = temp[c];
                        s++;
                        }
                if (s+i == k-1) {
                   fputs("solve error:  \tSingular matrix.  Column of zeros.\n",
                                stderr);                               30
                   exit(1);
                   }

                /* generates the new set of equations */
```

```
            for (r=i+1;r<k;r++) {
                    tempr = over(a[r][i], a[i][i]);
                    x[r] = minus(x[r], times(x[i], tempr));
                    for (c=i+1;c<k;c++) a[r][c] = minus(a[r][c],
                                                times(a[i][c], tempr));
                    }                                                    40
            }
        if (ccabs(a[k-1][k-1]) == 0.0) {
            fputs("solve error:  \tSingular matrix.  Last pivot is zero.\n",
                    stderr);
            exit(1);
            }

        /* back substitution */
        for (r=k-1; r>=0; r--) {
            v[r] = over(x[r], a[r][r]);                                  50
            for (c=r+1;c<k;c++) {
                    tempr = over(v[c], a[r][r]);
                    v[r] = minus(v[r], times(a[r][c], tempr));
                    }
/*          printf("v[%d] = (%g,%g)\n", r, re(v[r]), im(v[r])); */
            }
}
```

test.c

```
/* test program for function gp.
   Yanko Sheiretov          1/20/94          1/25/94                    */

#include "complex.h"
#include "objects.h"

main()
{
        static struct sensor s = {25, 100, 1.0e-3, 2.66e-11, 0.127, 0.24,
                56.72, 0.0, 6.2832e-2, 0.0, 0.0, {2.70e-11, 0.0}, {0.0, 0.0}};
        static struct layer l[2] = {{{8.854e-12, 0.0}, {0.0, 0.0}, 1000.0},
                                    {{8.854e-12, 0.0}, {0.0, 0.0}, 1e-3}};
        complex result;

        result = gp(2, l, &s);
        printf("gain = %g \tphase = %g\n", re(result), im(result));
}
```

testgp.c

```
#include "complex.h"
#include "objects.h"

complex gp(n, l, s)
        int n;
        struct layer *l;
        struct sensor *s;
{
        double a, b;
        a = re(l->bulk);                                        10
        b = im(l->bulk);
        return (cmplx(a*a+b*b, a*b));
}
```

H.6 Tools

complex.c

```
/* Functions operating on complex variables.
   Yanko Sheiretov          1/12/94          1/25/94          */

#include "complex.h"
#include <math.h>
#define MAXFLOATX 1.844674352e19

complex cmplx(real, imaginary)      /* makes a complex number */
        double real, imaginary;
{                                                                    10
        complex r;

        re(r) = real;
        im(r) = imaginary;
        return r;
}

double ccabs(z)
        complex z;
{                                                                    20
        double x, y, r;

        x = re(z);
        y = im(z);
        if(x < MAXFLOATX && y < MAXFLOATX) return sqrt(x*x + y*y);
        else {
                if (x > y) {
                        y /= x;
                        r = x*sqrt(1.0+y*y); }
                else {                                               30
                        x /= y;
                        r = y*sqrt(1.0+x*x); }
```

```
                return r; }
}

complex recip(x)              /* calculates the reciprocal of a  */
    complex x;                /* complex number                  */
{
  double y;

  y = ccabs(x);
  return (cmplx(re(x)/y/y, -im(x)/y/y));
}

complex plus(x, y)
        complex x, y;
{
        return (cmplx(re(x)+re(y),im(x)+im(y)));
}

complex times(x, y)
        complex x, y;
{
        return (cmplx(re(x)*re(y)-im(x)*im(y),re(x)*im(y)+im(x)*re(y)));
}

complex scale(x, y)
        complex x;
        double y;
{
        return (cmplx(re(x)*y,im(x)*y));
}

complex minus(x, y)
        complex x, y;
{
        return (cmplx(re(x)-re(y),im(x)-im(y)));
}
```

```
complex over(x, y)                                                    70
        complex x, y;
{
        return (times(x,recip(y)));
}

complex csq(x)
        complex x;
{
        return (times(x,x));
}                                                                     80
```

```
/* This function solves k equations with k unknowns and m result vectors
   using gaussian elimination. Format av=x; a[k][k]; x[k][m]; where x is
   used as both the source and target matrices.
   Yanko Sheiretov          1/21/94                                    */

/* Modified to work with complex numbers 3/24/94                       */

#include "complex.h"
#include "objects.h"
#include "matrix.h"                                                    10
#include <stdio.h>

void    ysolve(a, x, k, m)
        complex a[ML][ML], x[ML][ML];
        int     k, m;
{
        complex tempx[ML], tempa[ML], temp;
        int     r, c, i, j, s;

        if (k > ML || m > ML) {                                        20
                fputs("ysolve error:  \tMaximum matrix dimension exceeded.\n",
                        stderr);
                exit(); }

        for(i=0;i<k-1;i++) {
                /* handles the case with zero leading coefficient */
            s = 0;
              while (ccabs(a[i][i]) == 0.0 && s+i < k-1) {
                      for(j=0;j<m;j++) tempx[j] = x[i][j];
                      for(c=i;c<k;c++) tempa[c] = a[i][c];             30
                      for(r=i;r<k-1;r++) {
                              for(j=0;j<m;j++) x[r][j] = x[r+1][j];
                              for(c=i;c<k;c++) a[r][c] = a[r+1][c];
                              }
```

```
                    for(j=0;j<m;j++) x[k-1][j] = tempx[j];
                    for(c=i;c<k;c++) a[k-1][c] = tempa[c];
                    s++;
                    }
                if (s+i == k-1) {
                 fputs("ysolve error:  \tMatrix singular - column of zeroes.\n",
                        stderr);
                 exit(1);
                }
                /* generates the new set of equations */
                for (r=i+1;r<k;r++) {
                        temp = over(a[r][i], a[i][i]);
                        for(j=0;j<m;j++) x[r][j] = minus(x[r][j],
                                times(x[i][j], temp));
                        for (c=i+1;c<k;c++) a[r][c] = minus(a[r][c],
                                times(a[i][c], temp));
                        }
                }

        if (ccabs(a[k-1][k-1]) == 0.0) {
          fputs("ysolve error:  \tMatrix singular - last pivot is zero.\n",
                stderr);
          exit(1);
        }
        /* back substitution */
        for (r=k-1; r>=0; r--)
          for (j=0; j<m; j++) {
                x[r][j] = over(x[r][j], a[r][r]);
                temp = over(x[c][j], a[r][r]);
                for (c=r+1;c<k;c++) x[r][j] = minus(x[r][j],
                        times(a[r][c], temp));
                }
}

/* The following function inverts a matrix */
/* The source matrix is destroyed */
```

```
void    invert(a, x, k)
        complex a[ML][ML];              /* source matrix */
        complex x[ML][ML];              /* target matrix */
        int     k;
{
        int     i, j;

        if (k > ML) {
                fputs("invert error:  \tMatrix dimension exceeded\n", stderr);          80
                exit(); }

        /* Set up unity matrix */
        for(i=0;i<k;i++)
          for(j=0;j<k;j++) {
            re(x[i][j]) = i==j ? 1.0 : 0.0;
            im(x[i][j]) = 0.0;
            }

        ysolve (a, x, k, k);                                                          90
}

/* This multiplies matrices: c = ab; a[k][m]; b[m][n]; c[k][n] */
void mul(a, b, c, k, m, n)
    complex a[ML][ML], b[ML][ML];       /* source matrices */
    complex c[ML][ML];                  /* target matrix  */
    int k, m, n;
{
 int h, i, j;
                                                                                    100
 for (h=0; h<k; h++)
   for (j=0; j<n; j++) {
     re(c[h][j]) = im(c[h][j]) = 0.0;
     for (i=0; i<m; i++)
         c[h][j] = plus(c[h][j], times(a[h][i], b[i][j]));
   }
```

```
}

/* This multiplies a vector by a matrix: c = ab; a[k][k]; b[k]; c[k] */
void mulv(a, b, c, k)                                                    110
    complex a[ML][ML], b[ML];          /* source matrix and vector */
    complex c[ML];                     /* target vector */
    int k;
{
 int i, j;

 for (j=0; j<k; j++) {
  re(c[j]) = im(c[j]) = 0.0;
  for (i=0; i<k; i++)
   c[j] = plus(c[j], times(a[j][i], b[i]));                              120
 }
}
```

H.7 Input/Output

chipinfo.c

```
/* Used to get user input for the properties of the different sensors.
   Yanko Sheiretov        1/21/94        2/17/94                    */

#include "complex.h"
#include "objects.h"
#include <stdio.h>
#include <string.h>

void chipinfo(s)
        struct sensor *s;                                              10
{
        static   char f[5] = "%lf", name[40] = "/u/yanko/Tmp/";
        FILE *fp;
        char flag=0;
        int  len;

        len = strlen(name);
        do {
           fputs("Please enter the sensor name.  A \"-\" means stdin\n",
                                        stdout);                       20
           scanf("%s", name+len);
           if (name[len] == '-') {
                flag = 1;
                fp = stdin;
                }
           else if((fp = fopen(name, "r")) == NULL)
                printf("Can't read the sensor file %s, please try again...\n",
                                name);
           }
        while (fp == NULL);                                            30

        if (flag) fputs("\tspatial wavelength [m]:  lambda = ", stdout);
```

```
        fscanf(fp, f, &s->lambda);
        if (flag) fputs("\tsubstrate permittivity [F/m]:  eox = ", stdout);
        fscanf(fp, f, &s->eox);
        if (flag)
           fputs("\tnormalized substrate thickness []:  lambda/4h = ", stdout);
        fscanf(fp, f, &s->h);
        s->h = 0.25/s->h;
        if (flag)                                                          40
          fputs("\tnormalized interelectrode spacing []:  a/lambda = ", stdout);
        fscanf(fp, f, &s->g);
        if (flag)
           fputs("\tnormalized load capacitance []:  CL/(eox*ML) = ", stdout);
        fscanf(fp, f, &s->yload);
        if (flag) fputs("\tparasitic slope []:  ap = ", stdout);
        fscanf(fp, f, &s->ap);
        if (flag) fputs("\tparasitic intercept []:  bp = ", stdout);
        fscanf(fp, f, &s->bp);
        putc('\n', stdout);                                                50
        if (!flag) fclose(fp);
        name[len] = '\0';
}

void chipinfoout(s, fp)
        struct sensor *s;
        FILE    *fp;
{
        fprintf(fp, "\tspatial wavelength [m]:  lambda = %g\n", s->lambda);
        fprintf(fp, "\tsubstrate permittivity [F/m]:  eox = %g\n", s->eox);      60
        fprintf(fp, "\tnormalized substrate thickness []:  lambda/4h = %g\n",
                                0.25/s->h);
        fprintf(fp, "\tnormalized interelectrode spacing []:  a/lambda = %g\n",
                                s->g);
        fprintf(fp, "\tnormalized load capacitance []:  CL/(eox*ML) = %g\n",
                                s->yload);
        fprintf(fp, "\tparasitic slope []:  ap = %g\n", s->ap);
        fprintf(fp, "\tparasitic intercept []:  bp = %g\n", s->bp);
```

```
}
```

layinfo.c

```
/* Used to get user input for the properties of the different layers.
   Yanko Sheiretov          1/21/94          1/25/94                    */

#include "complex.h"
#include "objects.h"
#include <stdio.h>

void layinfo(l)
        struct layer *l;
{                                                                        10
        fputs("\tlayer thickness [m]:  ", stdout);
        scanf("%lf", &l->thickness);
        fputs("\tbulk permittivity and conductivity [F/m], [S/m]:  ", stdout);
        scanf("%lf,%lf", &re(l->bulk), &im(l->bulk));
        fputs("\tsurface permittivity and conductivity [F], [S]: ", stdout);
        scanf("%lf,%lf", &re(l->surface), &im(l->surface));
        putc('\n', stdout);
}

void layinfoout(l, fp)                                                   20
        struct layer *l;
        FILE *fp;
{
        fprintf(fp, "\tlayer thickness [m]:  %g\n", l->thickness);
        fprintf(fp,"\tbulk permittivity and conductivity [F/m],[S/m]:  %g,%g\n",
                        re(l->bulk), im(l->bulk));
        fprintf(fp, "\tsurface permittivity and conductivity [F],[S]: %g,%g\n",
                        re(l->surface), im(l->surface));
}
```

H.8 Sample Files

H.8.1 Input to Estimation Routines

Sample Input File for **est.c**	
out11	*Output file name*
0.01	*Frequency in* Hz
2.70e-11,0.0	*Bulk permittivity and conductivity below the electrodes*
0.0,0.0	*Surface permittivity and conductivity below the electrodes*
2	*Total number of layers*
2	*Number of unknown layers*
0,1	*Index numbers of the unknown layers*
mul25	*Name of template file for the zeroth sensor*
mul10	*Name of template file for the first sensor*
1000.0	*Thickness in meters of the zeroth layer*
8.854e-12,0.0	*Guesses for* ϵ *and* σ *for this layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
0.25e-3	*Thickness in meters of the first layer*
8.854e-12,0.0	*Guesses for* ϵ *and* σ *for this layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
-38.30,-43.68	*Gain in* dB *and phase in* deg *for the*
-41.40,-57.98	*two sensors.*

Sample Input File for **estm.c** and **ests.c**	
out11	*Output file name*
0.01	*Frequency in* Hz
2.70e-11,0.0	*Bulk permittivity and conductivity below the electrodes*
0.0,0.0	*Surface permittivity and conductivity below the electrodes*
2	*Total number of layers*
2	*Number of sensors*
2	*Number of unknown layers*
0,1	*Index numbers of the unknown layers*
mul25	*Name of template file for the zeroth sensor*
mul10	*Name of template file for the first sensor*
1000.0	*Thickness in meters of the zeroth layer*
8.854e-12,0.0	*Guesses for* ϵ *and* σ *for this layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
0.25e-3	*Thickness in meters of the first layer*
8.854e-12,0.0	*Guesses for* ϵ *and* σ *for this layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
-38.30,-43.68	*Gain in* dB *and phase in* deg *for the*
-41.40,-57.98	*two sensors.*

Sample Input File for **estp.c**	
out11	*Output file name*
0.01	*Frequency in* Hz
2.70e-11,0.0	*Bulk permittivity and conductivity below the electrodes*
0.0,0.0	*Surface permittivity and conductivity below the electrodes*
2	*Number of layers*
1	*Index number of the unknown layer*
10	*Number of sublayers*
mul50	*Name of template file for the zeroth sensor*
mul25	*Name of template file for the first sensor*
mul10	*Name of template file for the second sensor*
1000.0	*Thickness in meters of the zeroth layer*
8.854e-12,0.0	*Guesses for* ϵ *and* σ *for this layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
-0.7	*Logarithmic slope of decay for pressboard*
3.3	*Normalized* ϵ_∞*, i.e.* $\epsilon_\infty/\epsilon_0$
0.25e-3	*Thickness in meters of the first layer*
0.0,0.0	*Surface permittivity and conductivity for this layer*
1.0,-2.0	*Initial guess for the unknown complex parameter A*
2.0	*Initial guess for the unknown parameter B*
0.0	*Initial guess for the unknown parameter D*
-25.68,-72.72	*Gain in* dB *and phase in* deg *for the*
-16.78,-56.17	*three sensors.*
-20.63,-52.55	

H.8.2 Sensor Template Files

mul50

Template file for the longest wavelength	
5.0e-3	*Sensor wavelength in meters*
2.66e-11	*Substrate permittivity*
9.84	*Normalized substrate thickness* $\lambda/4h$
0.24	*Normalized interelectrode spacing* a/λ
0.81	*Normalized load capacitance* $C_L/\epsilon_{ox}M_L$
0.0	*Parasitic Slope*
-0.074	*Parasitic Intercept*

mul25

Template file for the medium wavelength	
2.5e-3	*Sensor wavelength in meters*
2.66e-11	*Substrate permittivity*
4.92	*Normalized substrate thickness* $\lambda/4h$
0.24	*Normalized interelectrode spacing* a/λ
9.42	*Normalized load capacitance* $C_L/\epsilon_{ox}M_L$
0.0	*Parasitic Slope*
-0.01	*Parasitic Intercept*

mul10

Template file for the small wavelength	
1.0e-3	*Sensor wavelength in meters*
2.66e-11	*Substrate permittivity*
1.97	*Normalized substrate thickness* $\lambda/4h$
0.24	*Normalized interelectrode spacing* a/λ
56.72	*Normalized load capacitance* $C_L/\epsilon_{ox}M_L$
0.0	*Parasitic Slope*
0.02	*Parasitic Intercept*

Bibliography

[1] A. K. Jonscher, *Dielectric Relaxation in Solids,* Chelsea Dielectrics Press, London, 1983.

[2] P. A. von Guggenberg, *Applications of Interdigital Dielectrometry to Moisture and Double Layer Measurements in Transformer Insulation,* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1993.

[3] M. C. Zaretsky, *Parameter Estimation Using Microdielectrometry with Application to Transformer Monitoring,* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November 1987.

[4] *CRC Handbook of Chemistry and Physics,* 61st edition, CRC Press, Inc., Boca Raton, FL, 33431, 1981.

[5] B. Nettelblad, *Effect of Moisture Content on the Dielectric Properties of Cellulose,* NORD-IS 92, Paper 8.9.

[6] U. Gäfvert, B. Nettelblad, *Measurement Techniques for Dielectric Response Characterization at Low Frequencies,* NORD-IS 92, Paper 7.1.

[7] M. C. Zaretsky, L. Mouayad, J. R. Melcher, *Continuum Properties from Interdigital Electrode Dielectrometry,* IEEE Transactions on Electrical Insulation, Vol. 23, No. 6, pp. 897–917, December 1988.

[8] F. B. Hildebrand, *Advanced Calculus for Applications,* 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[9] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes; The Art of Scientific Computing*, Cambridge University Press, 1986.

[10] W. McC. Siebert, *Circuits, Signals, and Systems,* The MIT Press, Cambridge, MA, 1986.

[11] P. Li, *Low Frequency, Millimeter Wavelength, Interdigital Dielectrometry of Insulating Media in a Transformer Environment*, LEES Technical Report TR-87-005, Massachusetts Institute of Technology, May, 1987.

[12] Y. Sheiretov, M. Zahn, *A Study of the Temperature and Moisture Dependent Dielectric Properties of Oil-Impregnated Pressboard*, 1993 Conference on Electrical Insulation and Dielectric Phenomena, Pocono Manor, PA, October 17–20, 1993.

[13] Y. Sheiretov, M. Zahn, *Dielectrometry Measurements of Moisture Dynamics in Oil-Impregnated Pressboard*, 1994 IEEE International Symposium on Electrical Insulation, Pittsburgh, PA, June 5–8, 1994.

[14] Y. Sheiretov, M. Zahn, *Dielectrometry Measurements of Spatial Moisture Profiles in Oil-Impregnated Pressboard*, 4th International Conference on Properties and Applications of Dielectric Materials, The University of Queensland, Brisbane, Australia, July 3–6, 1994.

[15] J. R. Melcher, *Continuum Electromechanics*, The MIT Press, Cambridge, MA, 1981.