AN ANALYSIS OF THE INTERFEROMETRIC ACOUSTO-OPTIC SPECTRUM ANALYZER

by

ALAN LEWIS FERGUSON

B.S., Kansas State University, 1986

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Donald R. Hummels

Major Professor

Table of Contents

List of Figures

List of Figures (continued)

Acknowledgments

I would like to thank Dr. Donald Hummels for his support and technical expertise throughout this project. Also, I would like to thank the Motorola Government Electronics Group for the financial support of the project. Thanks also go to my mother and father for their support through all my years of schooling.

I. INTRODUCTION

Acousto–optic (AO) spectrum analyzers have been found to be a very efficient device to perform real–time spectral analysis on signals. An AO spectrum analyzer uses the interaction between surface acoustic waves (SAW) and a light source to produce the spectrum of a signal. This report gives a mathematical basis for the interferometric acousto–optic (IFAO) spectrum analyzer and a computer algorithm to predict the system performance.

The development begins with work done by Vander Lugt [1] in analyzing the IFAO spectrum analyzer. The development is subsequently enlarged to make the mathematical model more flexible. This allows for more general input signal and reference waveforms. A computer algorithm is then developed, and some results obtained using the algorithm are presented and interpreted.

Interferometric Acousto–optic Spectrum Analyzers

The standard AO spectrum analyzer is shown in Figure 1. A monochromatic light source, usually a LASER, illuminates the analyzer. The signal to be analyzed, r(t), is input through a Bragg diffraction cell which modulates the light with the incoming signal. The light then passes through a lens which focuses the light beam at the plane of a photo–diode array. The amplitude of the light along the diode array is proportional to the Fourier transform of the input signal.

An IFAO spectrum analyzer differs slightly from the standard AO analyzer. In the interferometric analyzer, two light beam paths are used. The configuration of such an analyzer is shown in Figure 2. One of these paths illuminates a Bragg diffraction cell and is modulated by

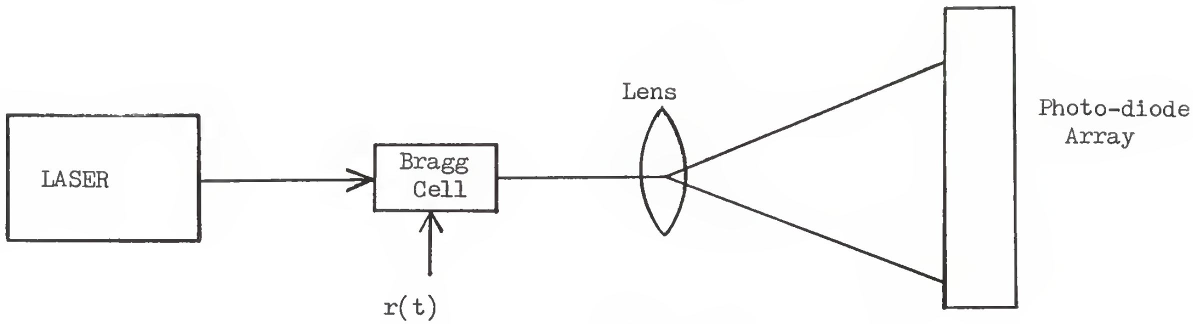

Figure 1. Conventional Acousto-optic Spectrum Analyzer

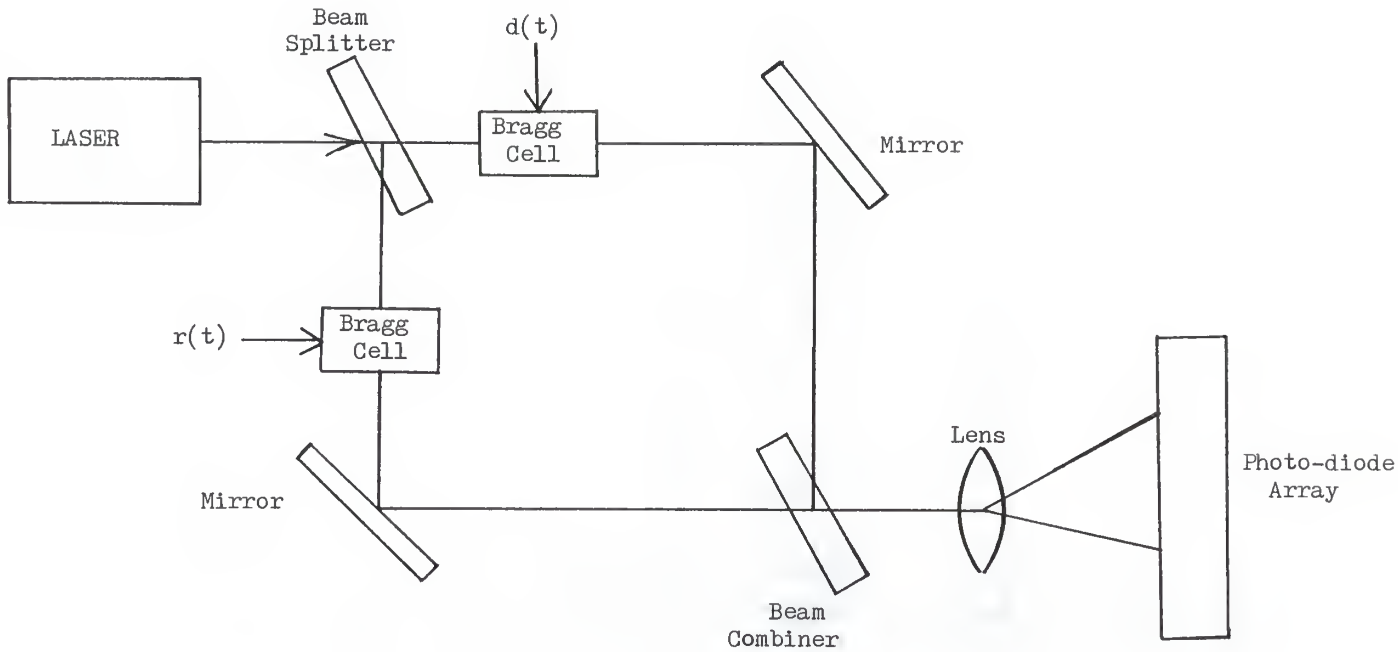

Figure 2. Interferometric Acousto-optic Spectrum Analyzer

the signal waveform, r(t). The other light beam path passes through another Bragg cell and is modulated by a reference waveform, d(t). The two modulated beams are then combined and passed through a lens which focuses the sum at the plane of a photo diode array. The light at the diode array depends on the interference pattern created by the two light sources, and hence the name "interferometric AO spectrum analyzer". The output signals of the diodes are then filtered and detected to provide an analog of the spectrum of r(t).

The IFAO spectrum analyzer has been proposed as being a device that will have extended dynamic range over the standard AO spectrum analyzer. Since photo-diode current is proportional to light intensity, conventional AO spectrum analyzers have an output that is proportional to the square of the input signal. Vander Lugt [1], in an analysis which is expanded in Section II, showed that when the optics are properly adjusted, the IFAO spectrum analyzer has photo-diode outputs which are linearly dependent on the input signal. The result is that the dynamic range of the hardware implementation of the analyzer, when expressed in dB, is essentially doubled with an IFAO approach.

II. A SYSTEM MODEL

An equivalent mathematical model for one diode channel of the IFAO spectrum analyzer is shown in Figure 3. The analysis will begin with the signal beam path, and a similar derivation will hold for the reference path. A signal, r(t), is applied to the Bragg cell. modulating the light that illuminates it. The light leaving the cell may be expressed as

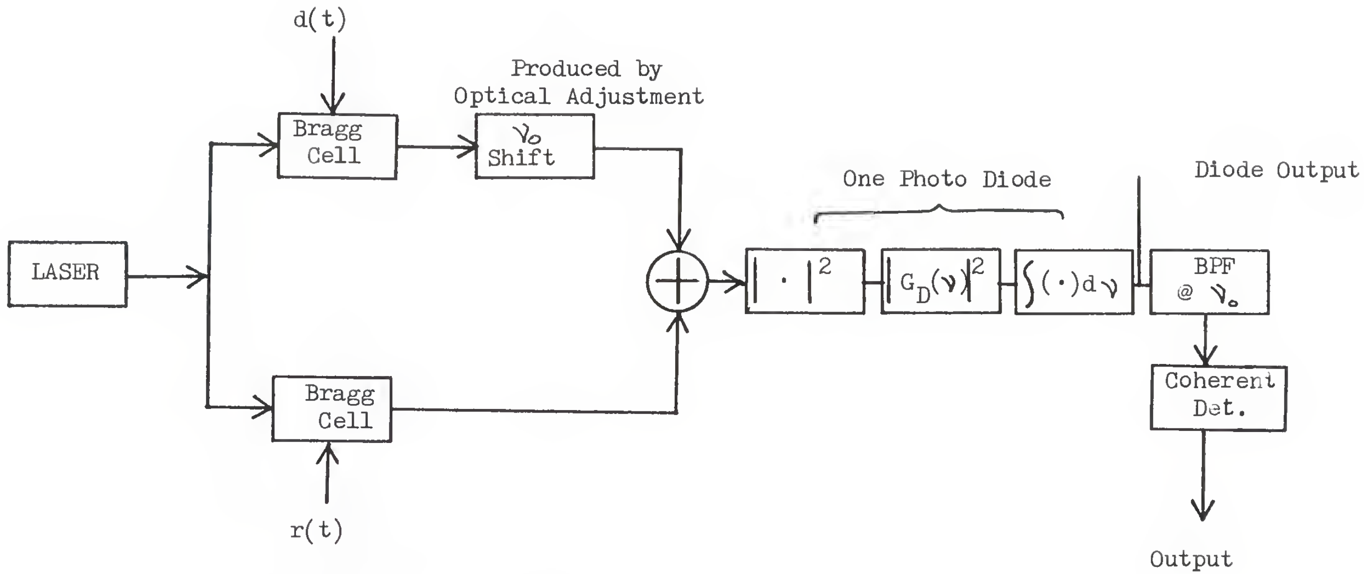

Figure 3. Equivalent Low-pass Model of IFAO with Coherent Detection

$$\mathrm{Re}\{\ E_c\ e^{j2\pi\nu t}\ e^{j\psi(x)}\ \} \tag{1}$$

where ν is the frequency of the light and

$$\psi(x) = \psi_o + \psi_m\, r(t - x/v). \tag{2}$$

In (2), ψ_o and ψ_m are constants, v is the velocity of the acoustic wave produced by r(t), and r(t) is the radio frequency signal for which a spectral analysis is desired. In general, r(t) may consist of signal s(t) plus random noise n(t).

A series representation for $e^{j\psi(x)}$ is

$$e^{j\psi(x)} = 1 + j\psi(x) + \ldots. \tag{3}$$

Normally the phase variations, $\psi(x)$, are small so that the first two terms of the series are a good approximation. Thus (1) is essentially equivalent to

$$\mathrm{Re}\{\ E_c\ e^{j2\pi\nu t}\ [1 + j\psi(x)]\ \}. \tag{4}$$

Substitution of (2) into (4) yields

$$\mathrm{Re}\{\ E_c\ e^{j2\pi\nu t}\ [1 + j\psi_o + j\psi_m r(t - x/v)]\ \}. \tag{5}$$

It is convenient to represent r(t) as

$$r(t) = \mathrm{Re}\{\ \tilde{r}(t)\ e^{j\omega_c t}\ \} \tag{6}$$

where $\tilde{r}(t)$ is the complex envelope of $r(t)$ and ω_c is the center or carrier frequency of $r(t)$. The first order term in (5) and the one of interest in this case is

$$\mathrm{Re}\{\ jE_c\ e^{j2\pi\nu t}\ \psi_m r(t - x/v)\ \}. \tag{7}$$

The complex amplitude of the light is

$$j\psi_m E_c r(t - x/v) = j\psi_m E_c \mathrm{Re}\{\ \tilde{r}(t - x/v)\ e^{j\omega_c (t-x/v)}\ \}. \tag{8}$$

The representation in (8) does not take into account any mask at the Bragg cell or any variation in the intensity of the light entering the cell. These effects may be taken into account by adding a window function $w(x)$ to (8) so that the complex amplitude of the light leaving the cell becomes

$$j\psi_m E_c\ w(x)\ \mathrm{Re}\{\ \tilde{r}(t - x/v)\ e^{j\omega_c (t-x/v)}\ \}. \tag{9}$$

The lens focuses the light at the plane of the photo diode array. The complex light amplitude at the focal plane, from the signal path is the Fourier transform [2] of (9). Specifically, we have

$$A_1(p,t) = C_1 \int_{-\infty}^{\infty} w(x)\left[\tilde{r}(t-x/v)e^{j\omega_c(t-x/v)} + \tilde{r}^*(t-x/v)e^{-j\omega_c(t-x/v)}\right] e^{-jpx}dx \tag{10}$$

where $C_1 = j\phi_m E_c/2$ and p is a radian spatial frequency variable given by $p = 2\pi\nu\xi/F$. Here, ξ is the actual spatial variable at the focal plane of the lens, and F is the focal length of the lens. Now make the change of variables

$$u = t - x/v,$$

with the result

$$\frac{A_1(p,t)}{C_1 v} = e^{-jpvt}\left\{\int_{-\infty}^{\infty} w(t-u)\ \tilde{r}(u)\ e^{+j2\pi(f_c + \frac{pv}{2\pi})u}\ du + \int_{-\infty}^{\infty} w(t-u)\ \tilde{r}^*(u)\ e^{-j2\pi(f_c - \frac{pv}{2\pi})u}\ du\right\}. \tag{11}$$

Fourier transform relations may be used to rearrange (11) into the form

$$\frac{A_1(p,t)}{C_1 v} = e^{-jpvt}\left\{\int_{-\infty}^{\infty} \tilde{R}(f) \int_{-\infty}^{\infty} w(t-u)\ e^{-j2\pi\left[-f - (f_c + \frac{pv}{2\pi})\right]u}\ du\ df + \int_{-\infty}^{\infty} \tilde{R}^*(-f) \int_{-\infty}^{\infty} w(t-u)\ e^{-j2\pi\left[-f + (f_c - \frac{pv}{2\pi})\right]u}\ du\ df\right\} \tag{12}$$

The cases of interest involve window functions w(x) which are real and even. Observing that the Fourier transform of w(t − u) is

$$W\left[-f + \left(f_c - \frac{pv}{2\pi}\right) \right] e^{-j2\pi\left[-f + \left(f_c - \frac{pv}{2\pi}\right) \right]t} \tag{13}$$

allows further simplification of (12). After substitution and some manipulation, we have

$$\frac{A_1(p,t)}{C_1 v} = e^{j2\pi f_c t} \int_{-\infty}^{\infty} \tilde{R}(f)\, W\left[-f - \left(f_c + \frac{pv}{2\pi}\right) \right] e^{j2\pi ft}\, df$$
$$+ \; e^{-j2\pi f_c t} \int_{-\infty}^{\infty} \tilde{R}^*(-f)\, W\left[-f + \left(f_c - \frac{pv}{2\pi}\right) \right] e^{j2\pi ft}\, df. \tag{14}$$

The advantage of this form is that the complex light amplitude is expressed in terms of the Fourier transforms of the signal envelope $\tilde{r}(t)$ and the window function $w(x)$. These transforms are readily computed using fast Fourier transform (FFT) methods.

A similar analysis can be carried out for the reference path. Let $d(t)$ represent the reference waveform of our choosing. There are a number of possible choices for $d(t)$. The main criteria to be satisfied are that $d(t)$ have a frequency spectrum that is essentially flat over operating frequency band of the analyzer and that $d(t)$ be easy to generate. One possible reference choice is to use a carrier at the center of the operating band that is modulated by a pseudonoise (PN) sequence that has a clock rate somewhat greater than the input bandwidth of the analyzer. For the analysis, it is not necessary to specify the reference waveform. Repeating the pattern of the earlier analysis of the signal path will show that the complex amplitude of the light at the focal plane of the lens, from the reference path is given by

$$\frac{A_2(p,t)}{C_2 v} = e^{j2\pi f_d t} \int_{-\infty}^{\infty} \tilde{D}(f)\, W\Big[-f - (f_d + \tfrac{pv}{2\pi}) \Big]\, e^{j2\pi ft}\, df$$
$$+\ e^{-j2\pi f_d t} \int_{-\infty}^{\infty} \tilde{D}^*(-f)\, W\Big[-f + (f_d - \tfrac{pv}{2\pi}) \Big]\, e^{j2\pi ft}\, df \qquad (15)$$

where $\tilde{D}(f)$ is the Fourier transform of the complex envelope, $\tilde{d}(t)$, of the reference signal, and f_d is the center frequency of the reference signal. Henceforth, we set $f_d = f_c$, and assume frequency differences are accounted for in the complex envelopes of the signal and reference. This is not essential and indeed is unlikely to be the case, but it does simplify the arguments that follow without loss of generality.

The total light at the plane of the photo diode array is the sum of the two sources. The complex amplitude is then the sum of $A_1(p,t)$ and $A_2(p,t)$. Since the diode current is proportional to the intensity of the light reaching the diode junction, the output of any one diode is proportional to

$$|\ A_1(p,t) + A_2(p,t)\ |^2 = |\ A_1(p,t)\ |^2 + 2\mathrm{Re}\{\ A_1(p,t)\ A_2^*(p,t)\ \}$$
$$+\ |\ A_2(p,t)\ |^2. \qquad (16)$$

Consider the terms on the right side of (16). The first may be written

$$|\ A_1(p,t)\ |^2 = \Big|\ e^{j2\pi f_c t}\ F^{-1}\Big\{ \tilde{R}(f)\ W[\ -f - (f_c + \tfrac{pv}{2\pi})] \Big\}$$
$$+\ e^{-j2\pi f_c t}\ F^{-1}\Big\{ \tilde{R}^*(-f)\ W[\ -f + (f_c - \tfrac{pv}{2\pi})]\Big\}\Big|^2 \qquad (17)$$

where $F^{-1}\{\cdot\}$ denotes the inverse Fourier transform. This term represents the output of a conventional AO spectrum analyzer.

Completing the indicated square will result in a baseband term proportional to the square of a windowed version of the signal envelope and a similar term centered on 2 f_c Hz. Neither term is of interest here since they are more readily obtained in a conventional AO receiver, and they lack the desired linear dependence on the signal envelope $\tilde{r}(t)$. The same arguments hold for the term $| A_2(p,t) |^2$. Our attention focuses now on the product term $2\mathrm{Re}\{ A_1(p,t)\ A_2^*(p,t) \}$, as it should have the desired linear dependence on the signal.

A difficulty that may not be evident at this point remains to be overcome. The temporal frequencies contained in the cross product term overlap with those in the square terms, so that the terms may not be isolated in the present form. Vander Lugt solved this problem by adjusting the optics in the reference path, so that the spatial frequency of the light represented by $A_2(p,t)$ was shifted slightly. The effect was a corresponding shift in the temporal frequency of the cross product which allowed the desired linear term to be isolated. We proceed by replacing p by $p + p_0$ in $A_2(p,t)$ to account for the optical adjustment and form the cross product term as

$$\frac{A_1(p,t)\ A_2^*(p+p_0,t)}{C_1 C_2^* v^2} =$$

$$F^{-1}\Big\{ \tilde{R}(f)\ W[\ f + (f_c + \tfrac{pv}{2\pi})\ ]\ \Big\}\ F^{-1}\Big\{ \tilde{D}(f)\ W[\ f + (f_c + \frac{p+p_0}{2\pi}\ v)\ ]\ \Big\}^*$$
$$+\ F^{-1}\Big\{ \tilde{R}^*(-f)\ W[\ f - (f_c - \tfrac{pv}{2\pi})\ ]\ \Big\}\ F^{-1}\Big\{ \tilde{D}^*(-f)\ W[\ f - (f_c - \frac{p+p_0}{2\pi}\ v)\ ]\ \Big\}^* \tag{18}$$

where a term with temporal frequencies around 2 f_c Hz has been discarded. Since $\tilde{R}(f)$, $\tilde{D}(f)$ and $W(f)$ are all low pass functions, it is evident that the terms in (18) are disjoint with regard to the spatial frequency variable, p. Numerical solution is made simpler by the substitutions,

$$\nu_1 = f_c + \frac{pv}{2\pi}, \tag{19}$$

$$\nu_2 = f_c - \frac{pv}{2\pi}, \tag{20}$$

and

$$\nu_0 = \frac{p_0 v}{2\pi}. \tag{21}$$

Equation (18) then becomes

$$\frac{A_1(\nu,t)\ A_2^*(\nu+\nu_0,t)}{C_1 C_2^* v^2} =$$
$$F^{-1}\left\{ \tilde{R}(f)\ W(f+\nu_1) \right\}\ F^{-1}\left\{ \tilde{D}(f)\ W(f+\nu_1+\nu_0) \right\}^*$$
$$+\ F^{-1}\left\{ \tilde{R}^*(-f)\ W(f-\nu_2) \right\}\ F^{-1}\left\{ \tilde{D}^*(-f)\ W(f-\nu_2+\nu_0) \right\}^* \tag{22}$$

The output of a diode having an aperture centered on ν_1, $\nu_2 = 0$, or equivalently a spatial frequency of f_c Hz may now be written by integrating (22) over the appropriate aperture function,

$$e_d(t) =$$

$$G_0 \int_{-\infty}^{\infty} |G_d(\nu_1)|^2 \, \mathrm{Re}\left[F^{-1}\left\{\tilde{R}(f) \, W(f + \nu_1)\right\} F^{-1}\left\{\tilde{D}(f) \, W(f + \nu_1 + \nu_0)\right\}^*\right] d\nu_1 +$$

$$G_0 \int_{-\infty}^{\infty} |G_d(\nu_2)|^2 \, \mathrm{Re}\left[F^{-1}\left\{\tilde{R}^*(-f) \, W(f - \nu_2)\right\} F^{-1}\left\{\tilde{D}^*(-f) \, W(f - \nu_2 + \nu_0)\right\}^*\right] d\nu_2 . \tag{23}$$

The factor G_0 is a gain constant which depends on C_1, C_2 and v, and the aperture characteristic is denoted as $| \, G_d(\nu) \, |^2$.

It was determined by Vander Lugt [1] that when d(t) is chosen correctly, a component in $e_d(t)$ is sinusoidal with frequency ν_0 Hz and has amplitude linearly proportional to the spectrum of r(t) at frequency f_c. The signal $e_d(t)$ will have baseband terms, which have been discarded and are not evident in (23). It is necessary to filter off the component at ν_0 Hz and detect its amplitude to obtain the output of the spectrum analyzer. The filter must be narrow enough to filter out the unwanted terms.

A detector with a linear characteristic is needed to detect the signal. This ensures that the extended range of the interferometric spectrum analyzer is utilized. Using something other than a linear detection scheme, such as square-law detection, would nullify any gains made. Two types of detection are most likely, coherent detection and the use of a log video detector [3].

Coherent detection is assumed for this analysis, where the necessary reference signal at ν_0 Hz is obtained by injecting a low-level sine wave into the signal path at the edge of the analyzer input band. Taking the impulse response of the output filter to be h(t), the output from one diode in the interferometric AO spectrum analyzer may be written in the form

$$e_0(t) = \left[\int_{-\infty}^{\infty} e_d(\tau)\ h(t-\tau)\ d\tau \right] \cos(2\pi\nu_0 t + \phi) \tag{24}$$

where ϕ is a phase constant of the detector reference and is adjusted for maximum output. Since the phase of the signal is not known in advance, practical implementations will require two detectors with in-phase and quadrature references.

Although the extensive use of the Fourier transform simplifies the computer algorithm, it does not provide for much intuition regarding the operation of the IFAO spectrum analyzer. The following derivation attempts to make the operation more clear.

Only the first term of $e_d(t)$, as given in (23), is considered. A similar derivation will hold for the second term. As a model for a very narrow aperture, we assume that the diode aperture function, $|G_d(\nu_1)|^2$, is an impulse function at $\nu_1 = 0$. Then, the first term becomes

$$G_0\ \mathrm{Re}\left\{ F^{-1}\left\{ \tilde{R}(f)\ W(f) \right\}\ F^{-1}\left\{ \tilde{D}(f)\ W(f + \nu_0) \right\}^* \right\}. \tag{25}$$

Conjugation and taking inverse Fourier transforms yield

$$G_0\ \mathrm{Re}\left\{ \left\{ \tilde{r}(t) * w(t) \right\} \cdot \left\{ \tilde{d}^*(t) * \left[w(t)\ e^{+j2\pi\ \nu_0 t} \right] \right\} \right\}. \tag{26}$$

From this it can be seen that the action of the Bragg cell is to convolve the input with the window function corresponding to the particular cell. To get any output, both the signal waveform and the reference waveform must be non-zero simultaneously. Therefore it is important that the reference signal illuminate each diode in the array and that the illumination of the diodes be equal and constant over time. Equation (26) shows the large effect that the reference has upon the output of the analyzer.

Equations (23) and (24) provide a sufficient mathematical model for the IFAO spectrum analyzer. From this model a computer algorithm may be constructed to compute a variety of waveforms and spectra for the analyzer.

A Computer Algorithm

The computer algorithm was developed from the mathematical model just described. Extensive use of the fast Fourier transform (FFT) is made throughout the algorithm. An outline of the algorithm is as follows:

1. Initialize or input the following for the case of interest:

   $\tilde{r}(t)$ → signal waveform in time domain

   $\tilde{d}(t)$ → reference waveform in time domain

   $W(f)$ → Bragg Cell window in frequency domain

   $|G_d(f)|^2$ → diode aperture function

   $H(f)$ → frequency response of BP filter.

2. Compute $\tilde{R}(f)$ and form $\tilde{R}(f)\ W(f + \nu_1)$.

3. Compute $\tilde{D}(f)$ and form $\tilde{D}(f)\ W(f + \nu_1 + \nu_0)$.

4. Compute the appropriate inverse transforms and form:

$$\mathrm{Re}\left[F^{-1}\left\{ \tilde{R}(f)\ W(f + \nu_1) \right\}\ F^{-1}\left\{ \tilde{D}(f)\ W(f + \nu_1 + \nu_0) \right\}^* \right] \quad (27)$$

5. Multiply the result of Step 4 by $G_0\ |\ G_d(f)\ |^2$.

6. Repeat Steps 2 through 5 for a sequence of values of ν_1 where $|G_d(\nu_1)|^2$ is significant. Sum the results each time and multiply by the step size, $\Delta\nu$, to obtain a numerical integration of the first integral in (23).

7. Repeat steps 2 through 6 for the second integral in (23) and then sum the two to obtain $e_d(t)$.

8. Filter the signal with a bandpass filter centered on ν_0 Hz. To do this, first form the transform $E_d(f)$ and then multiply by the filter transfer function.

9. Perform coherent detection of the signal to obtain the envelope of the output signal. See Figure 4 for block diagram of the algorithm coherent detector.

The algorithm described was implemented in the C computer language and runs on a Digital Equipment VAX 11/750 computer. To provide the needed frequency resolution, 1024 data points are used. This implementation takes about 35 minutes of CPU time to execute the program. Several data files are produced by the program to monitor the performance at different points in the IFAO spectrum analyzer structure.

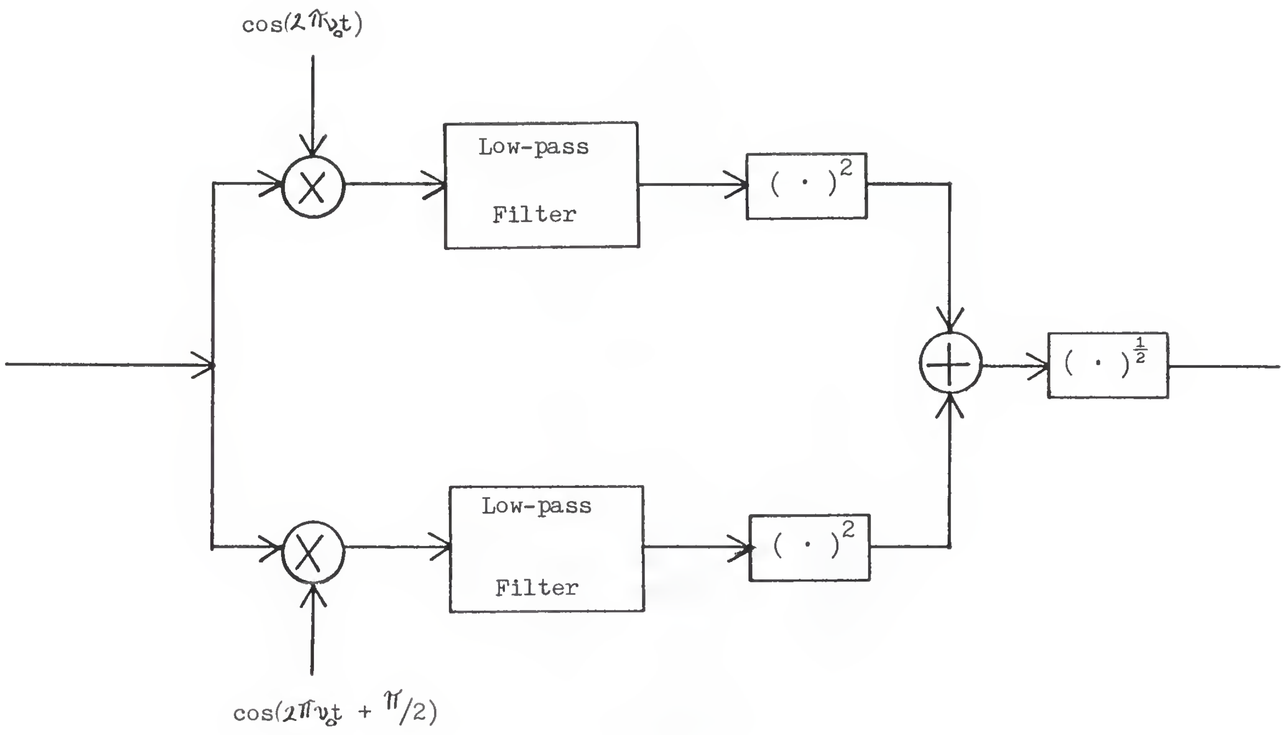

Figure 4. Coherent Detector Implementation

III. SUMMARY OF RESULTS

As currently written, the computer program requires the user to provide several inputs. Among these are parameters of input signal, the type of reference signal and the periods and sample times of analyzer waveforms. Also, the type of Bragg cell window and type of bandpass filter need to be defined. The parameters and waveforms used in many of the examples are described in the following paragraphs.

Signal Waveforms

Since the computer program used an equivalent lowpass model in the algorithm, any complex envelope which corresponds to a real signal could be used. It was decided to use a radar pulse for analysis. The case of zero rise and fall time was used; thus $\tilde{r}(t)$ is a rectangular pulse (see Figures 5 and 6 for the time and frequency domain representations). Such signals may also be delayed or changed to represent delayed pulses or different pulse widths.

Reference Waveforms

Four reference waveforms were used in the analysis: an impulse, a carrier modulated with pseudo-random noise, a reference with saw-tooth frequency sweep and a sum of sinusoids. The time and frequency domain representations for these signals can be seen in Figures 7 to 12. All four waveforms met the desired characteristic of having relatively flat frequency spectra in the range of interest and were easy to generate. The impulse reference was included for analytical use only and would not be a possible reference signal.

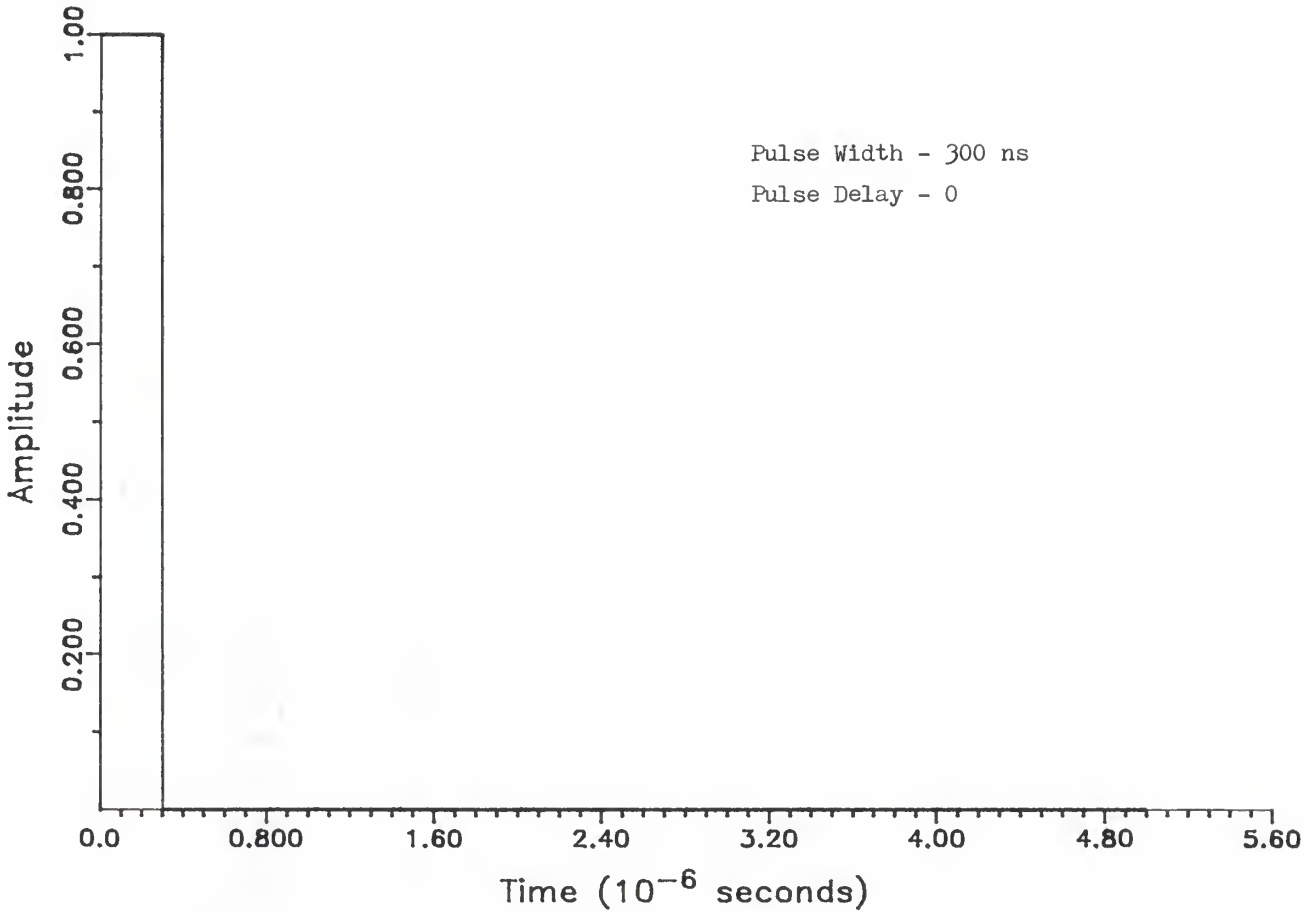

Figure 5. Rectangular Pulse in Time Domain

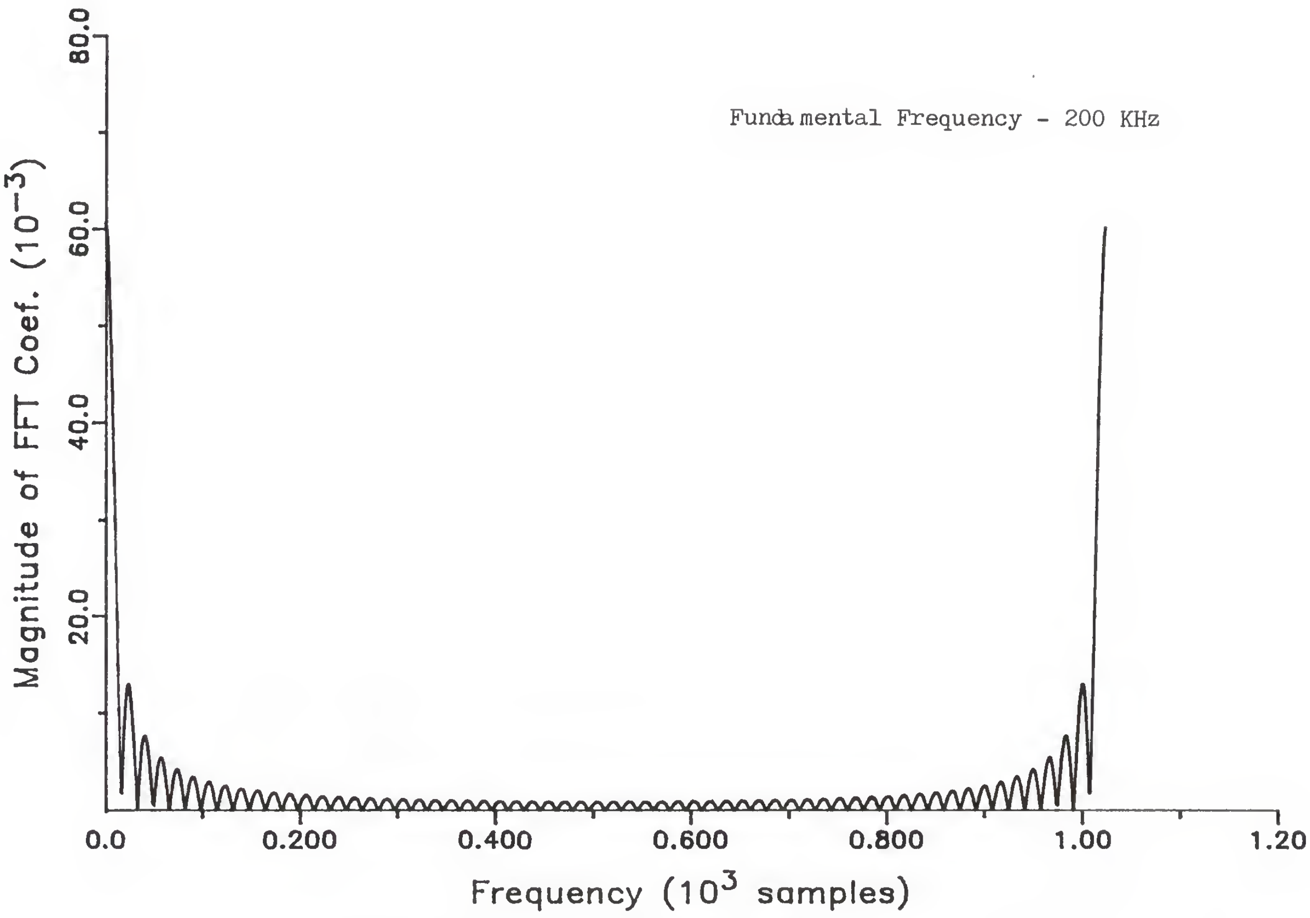

Figure 6. FFT of Rectangular Pulse

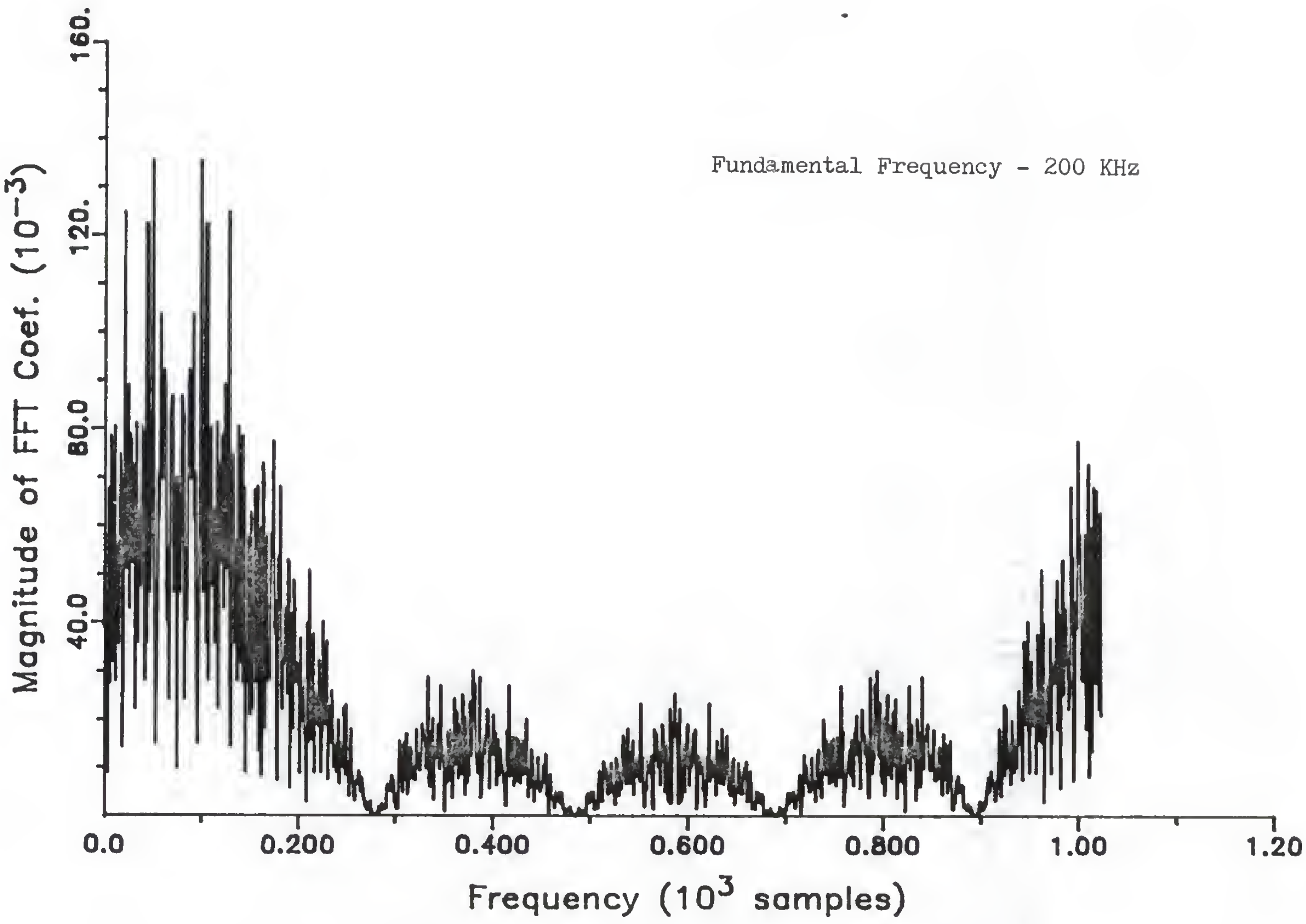

Figure 8. FFT of FM Reference

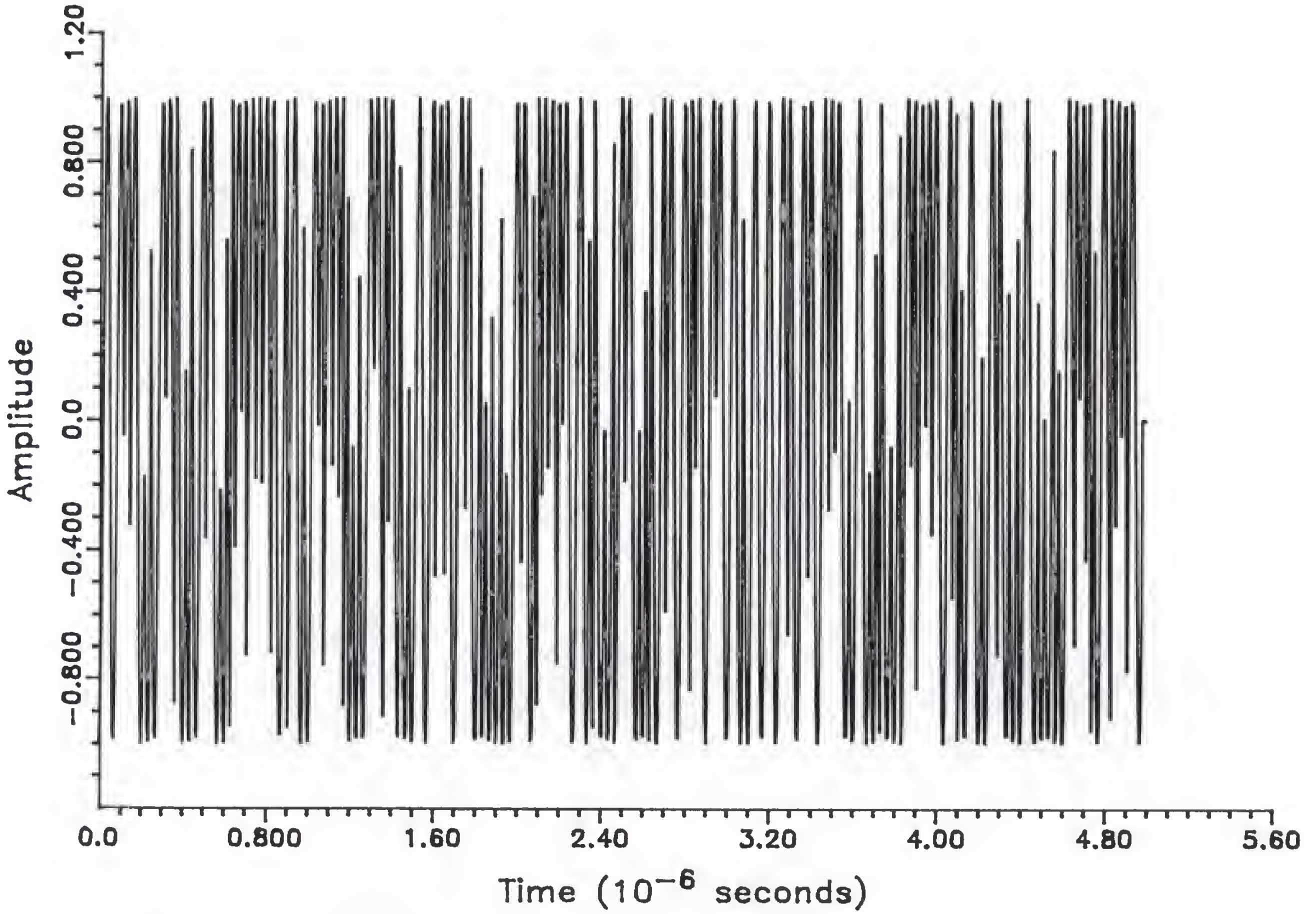

Figure 7. PN Reference in Time Domain

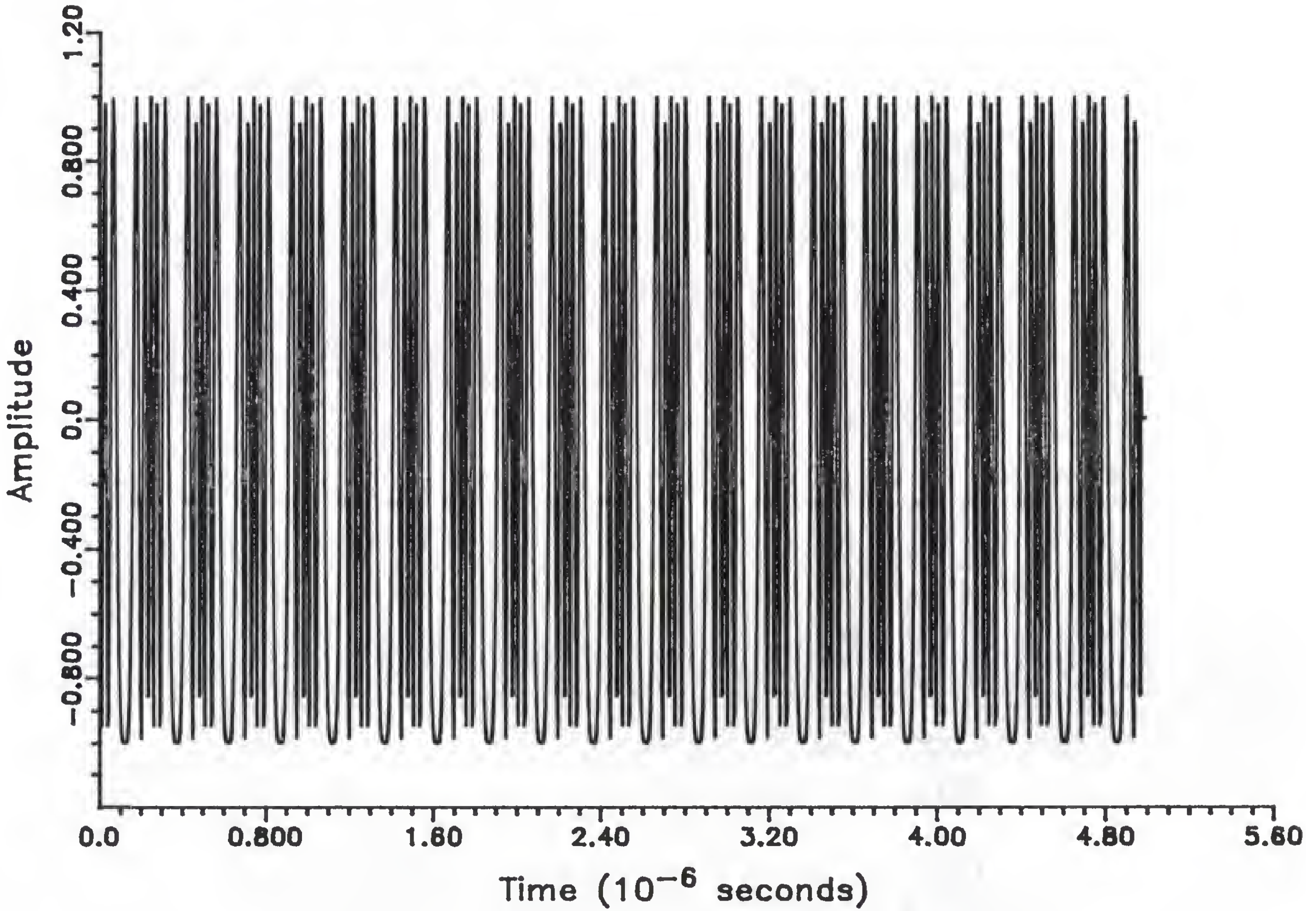

Figure 9. Frequency Sweep Reference in Time Domain

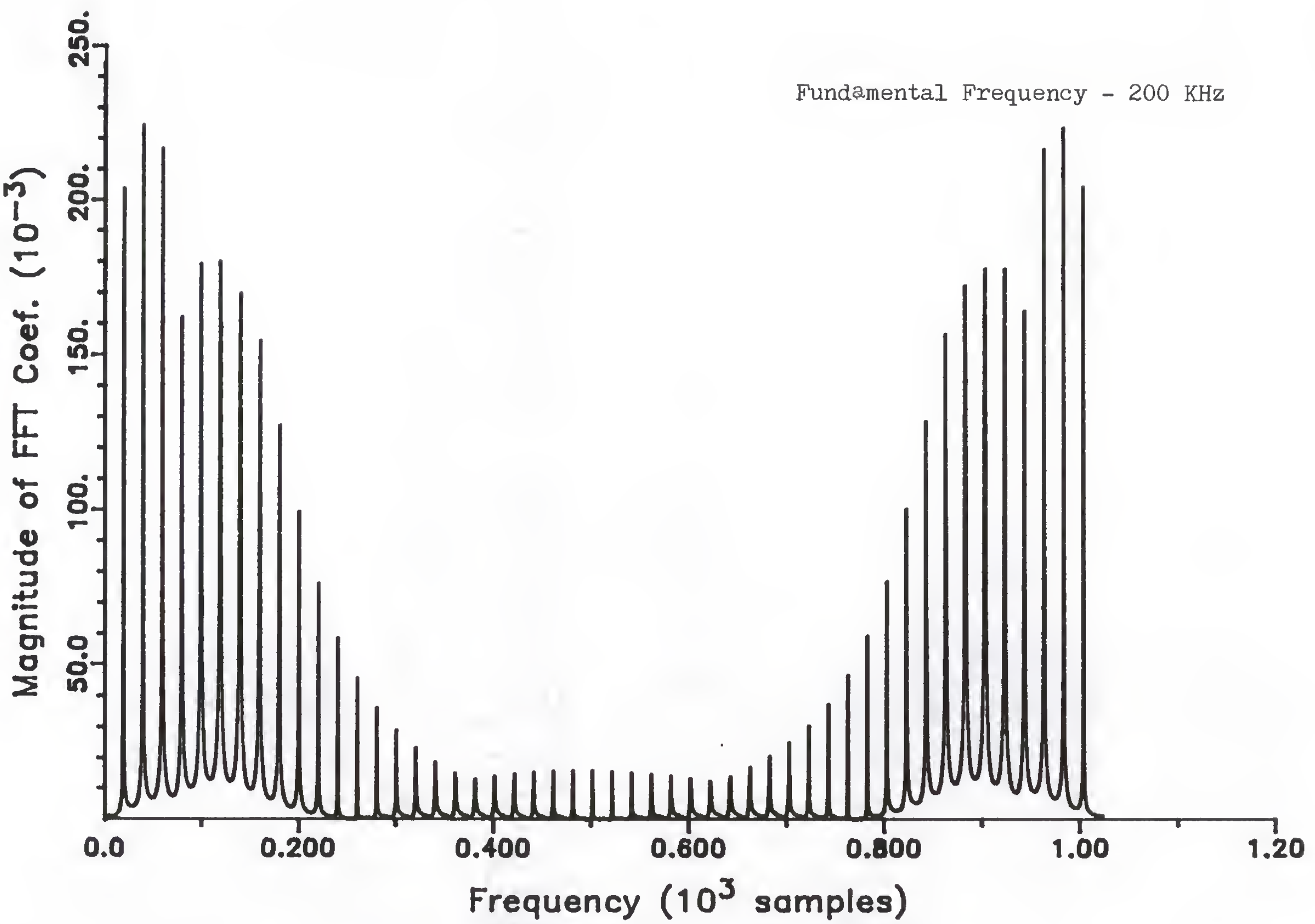

Figure 10. FFT of Frequency Sweep Reference

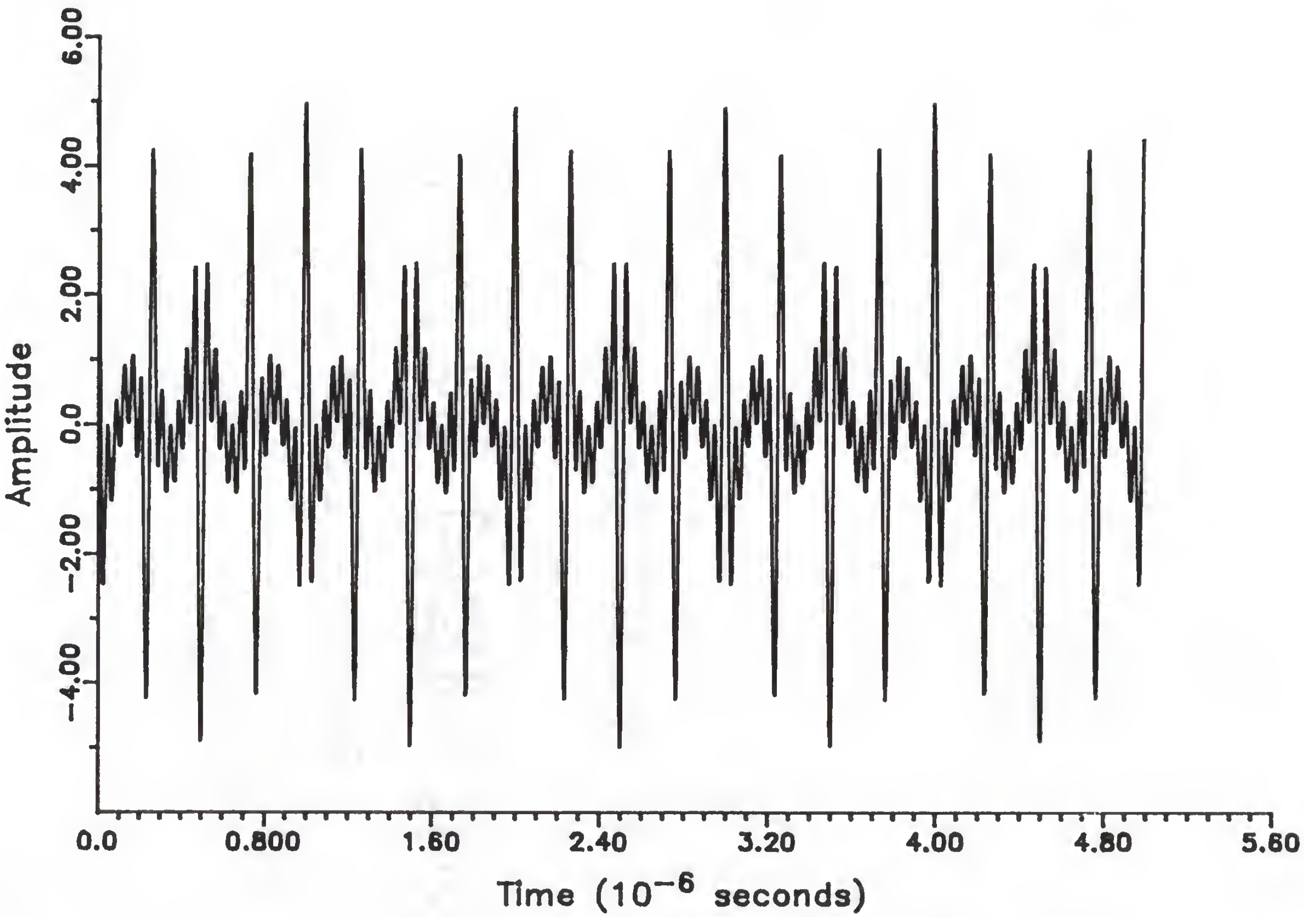

Figure 11. Sum of Sinusoids Reference in Time Domain

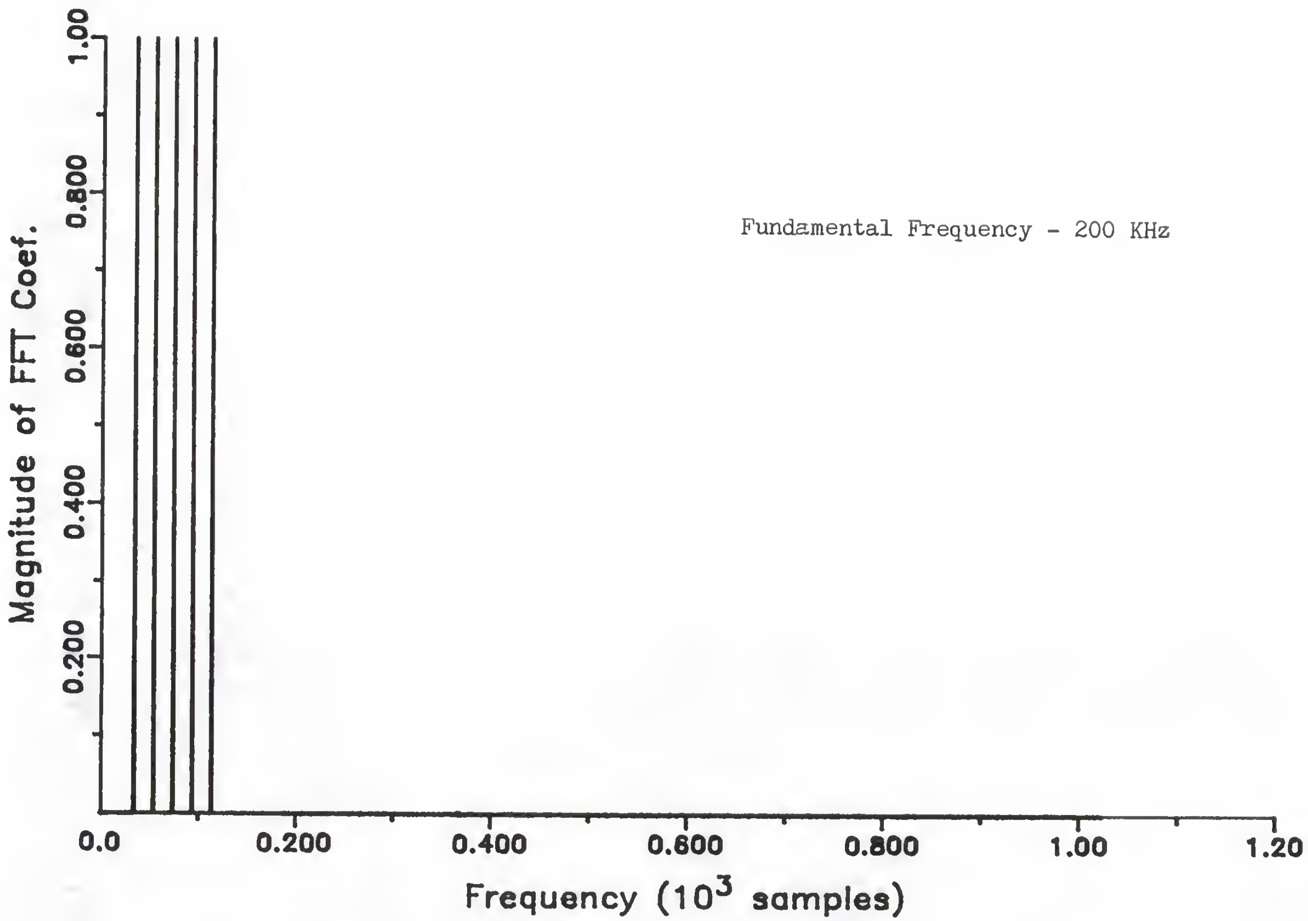

Figure 12. FFT of Sum of Sinusoids Reference

The PN sequence was generated by using a shift register PN sequence generator with a sequence length of 511 bits before repeating. Figure 13 shows the particular implementation. Each bit was held for 5 time samples to prevent aliasing problems. The bit was then used to change the phase of the reference carrier by 0 or π radians according to whether the bit was zero or one. A linear sweep was used for the frequency sweep reference. The reference sweeps the frequencies of –40 MHz to 40 MHz about f_c twenty times over the period of the input waveform. This ensures that the reference is continually sweeping through the frequencies contained in the input pulse.

The sum of sinusoids reference is a sum of five sinusoids centered at ν_0 spaced 4 MHz apart. Only the sinewave at frequency ν_0 is needed as reference signal for the channel of interest. The others would be necessary for adjacent diodes in the IFAO spectrum analyzer and are included to provide a check on adjacent channel interference.

Bragg Cell Window and Filtering

Two types of Bragg cell windows are implemented in the program: the Gaussian window and the rectangular window. It is the frequency domain representation of these windows that is of most interest. Figure 14 shows the differences in the two windows. The Gaussian window is obtained from

$$W(f) = \frac{\sqrt{\pi}}{\sigma} e^{-\pi^2 f^2/\sigma^2} \tag{28}$$

where $\sigma^2 = 36.966 \times 10^{12}$ sec^{-2}. This equation is found in [4] and has

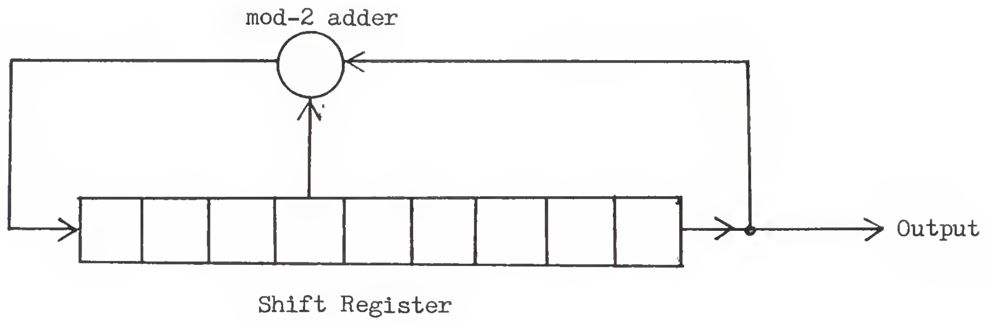

Figure 13. Shift Register for Generation of PN Sequence

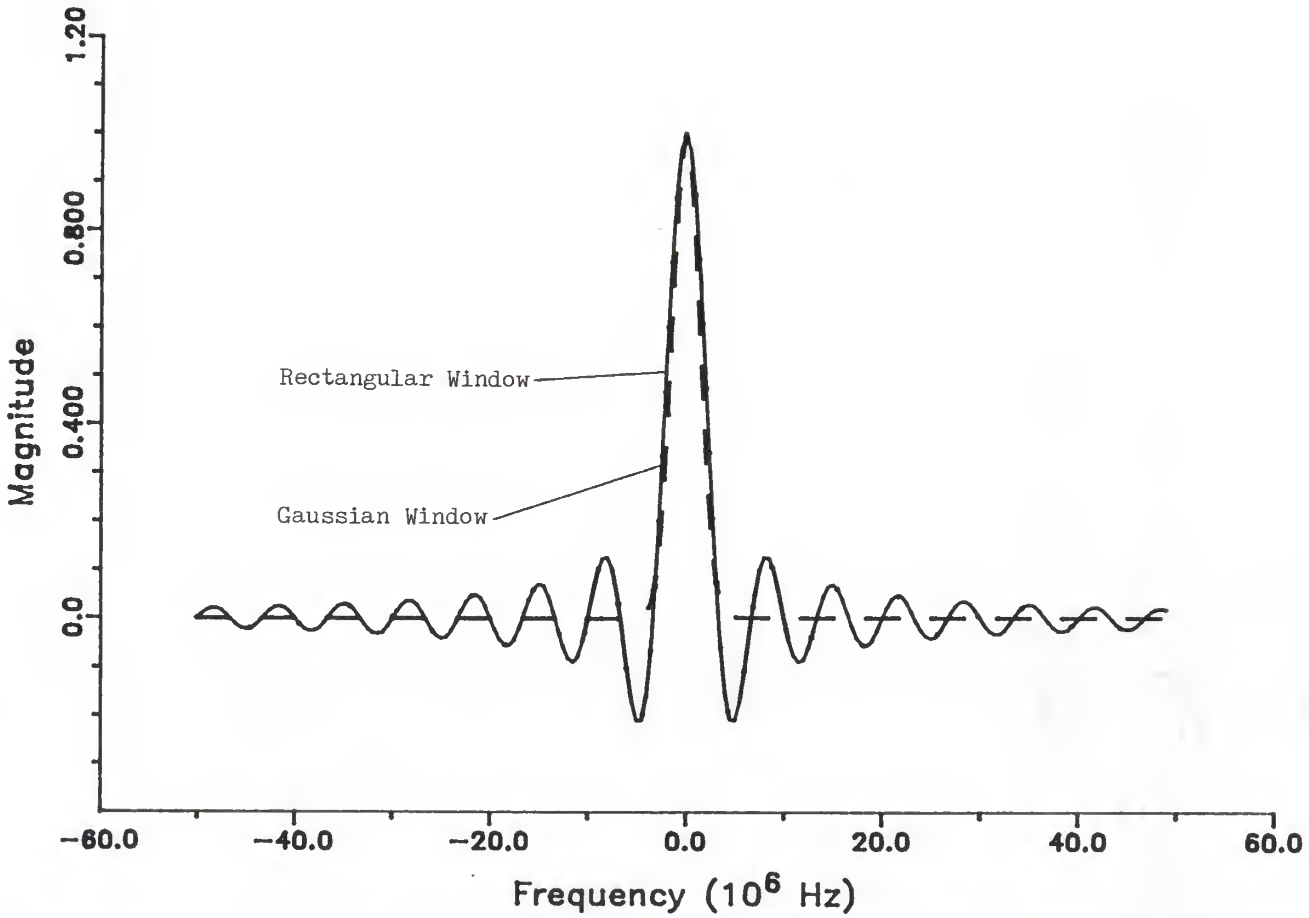

Figure 14. Bragg Cell Windows in Frequency Domain

been modified to represent a 250 ns Bragg cell window. The rectangular window assumes a time of 250 ns for the acoustic signal to propagate across the Bragg cell. Note that the two are similar except in the tails of the windows, where the (sin x)/x function shows up.

The type of diode aperture used for the results given herein was also found in [4]. It has a trapezoidal shape with physical dimensions that translate to a plateau width of 2.5 MHz and a 3 dB bandwidth of 5 MHz. This is shown in Figure 15. It is important to note that the equivalent bandwidths of the Bragg cell window and the diode aperture have a large effect on the output.

To simplify the program and the subsequent analysis, a one-pole bandpass filter was used to filter the diode output signal. Such a filter would not exhibit the ringing that higher order filters would have and allows for easy interpretation of the results.

Testing the Algorithm

To fully test both the algorithm and the computer implementation, several test cases were developed and compared. The parameters varied were:

Spatial Frequency Shift ν_0 – 15 MHz, 20 MHz, 30 MHz

Type of Reference – Impulse, Pseudo-random Noise, Frequency Sweep, Sum of Sinusoids

Type of Bragg Cell Window – Gaussian or Rectangular

Carrier Frequency Offset – 0, 1, 3 MHz

Pulse Width – 100, 200, 300, 400 ns

Pulse Delay – 0, 1, 2, 3, 4 μs

From these test cases, the effect of different parameters on the IFAO

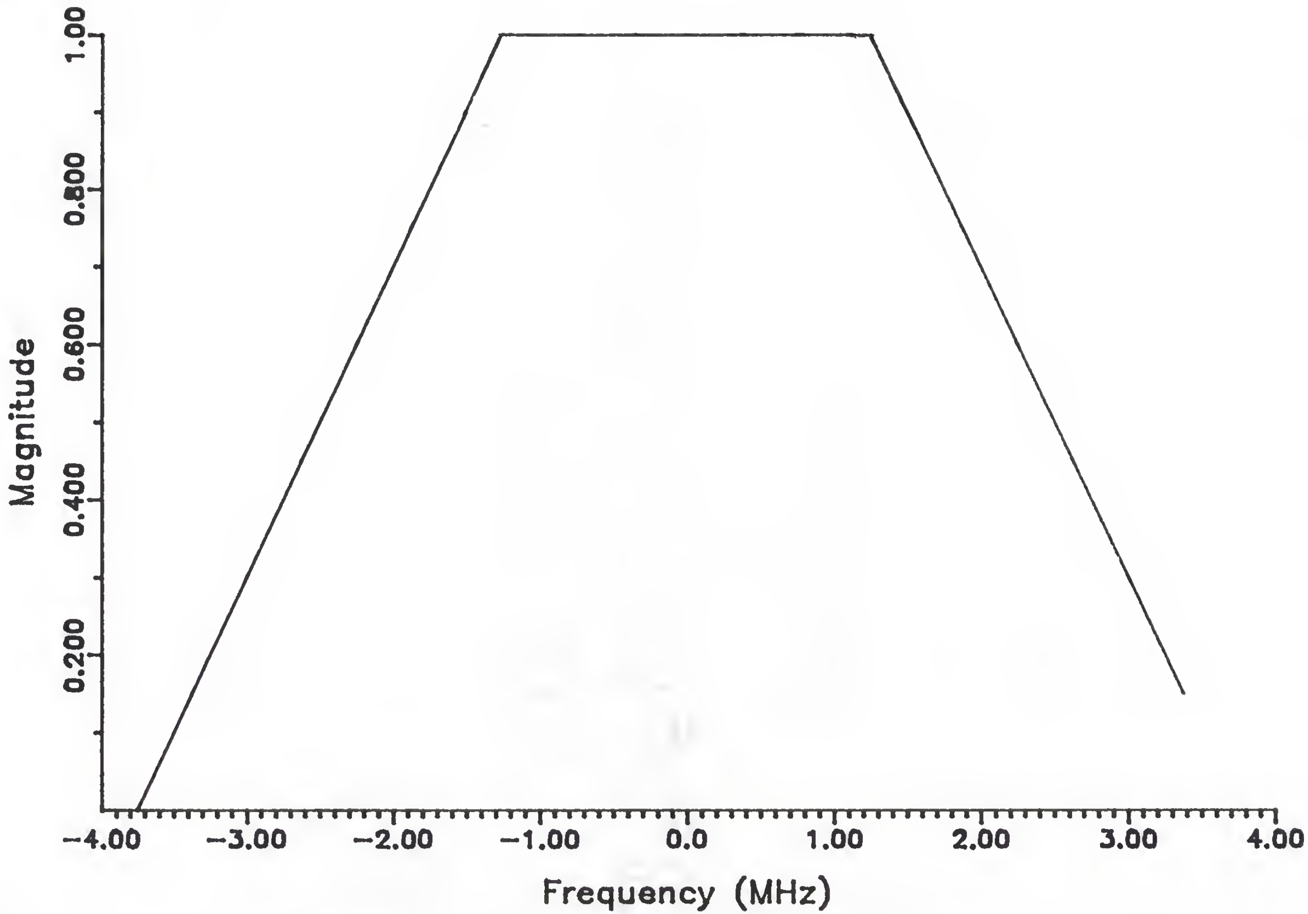

Figure 15. Diode Aperture in Frequency Domain

spectrum analyzer could be determined. The effects are summarized in the following sections. The case of a 300 ns rectangular pulse with a Gaussian aperture and PN reference is often used for comparison. Figures 16 through 19 show the response for these conditions at several places within the spectrum analyzer.

The diode output in the time domain is shown in Figure 16. Note that the envelope of the sinusoid appears to be the convolution of the pulse and the window. The frequency of the sinusoid is ν_0 (15 MHz), as can be better seen in Figure 17. Figures 18 and 19 show the resulting output of the filtering and coherent detection operations. Comparison of the time response of Figure 18 with the diode output of Figure 16 shows the former to be the envelope of the latter. These were the expected outputs of an IFAO spectrum analyzer with coherent detection.

Spatial Frequency Shift ν_0

Changing the value of ν_0 changes the position in the spatial dimension that the spectrum appears. In the hardware implementation this is done by adjusting the optics. Changing the length of the reference path will change the temporal frequency center ν_0. The result is best seen in Figure 20, where the frequency domain representation of the diode output signal is shown. Figure 21 displays the difference in output from the various choices of ν_0. The difference in amplitudes is due to the variations in the PN sequence for the different cases.

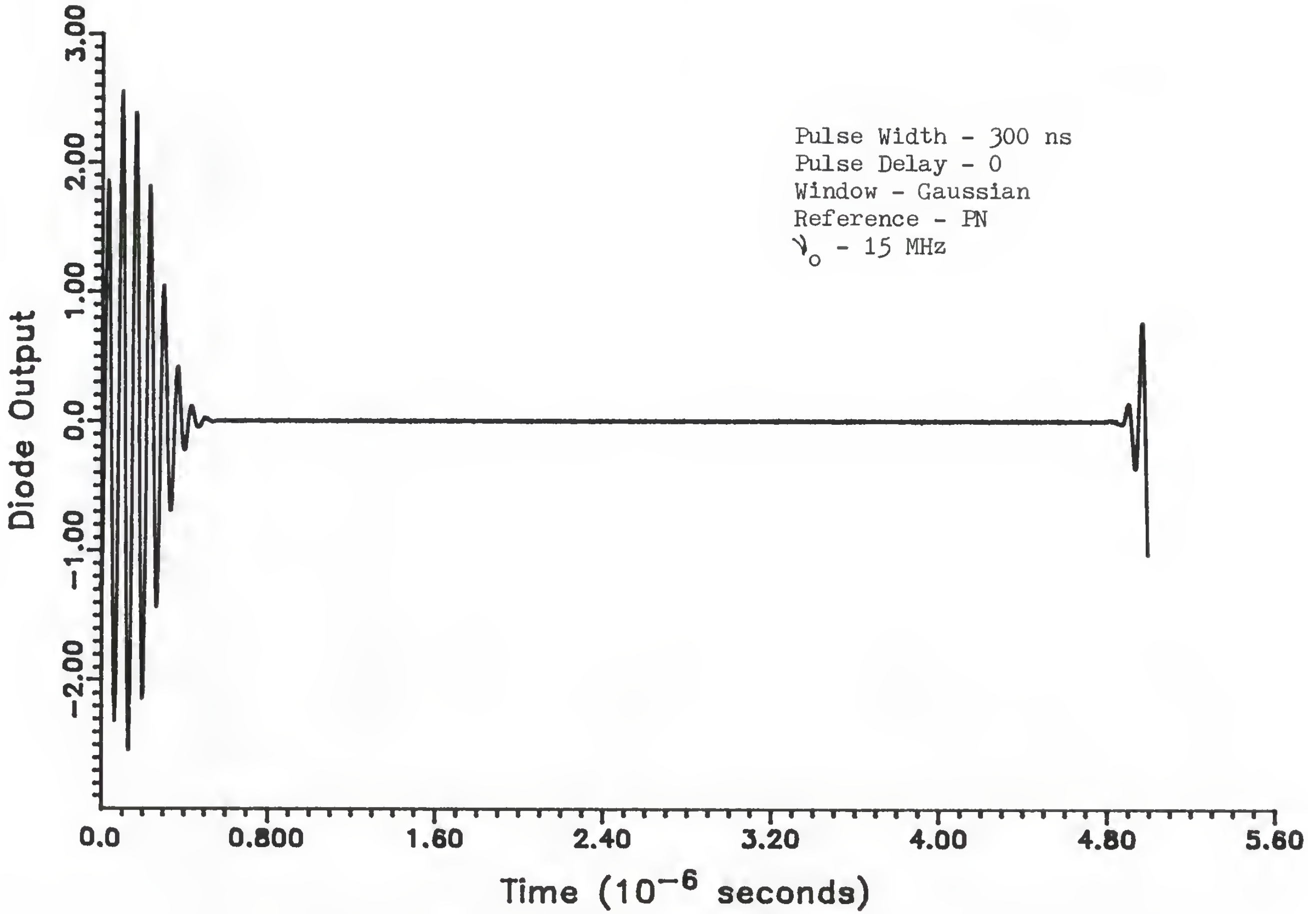

Figure 16. Diode Output in Time Domain, Rectangular Pulse Input

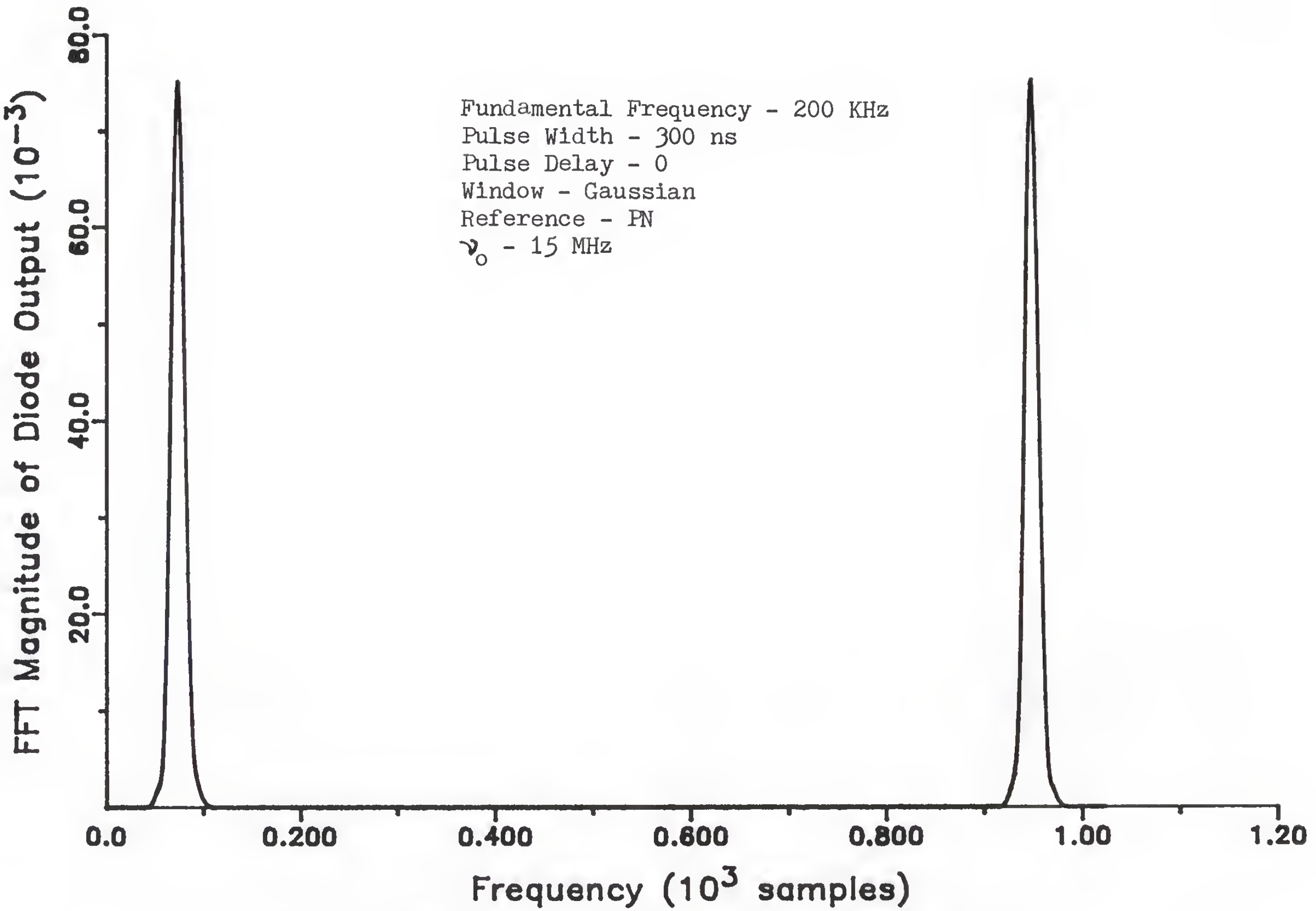

Figure 17. FFT of Diode Output, Rectangular Pulse Input

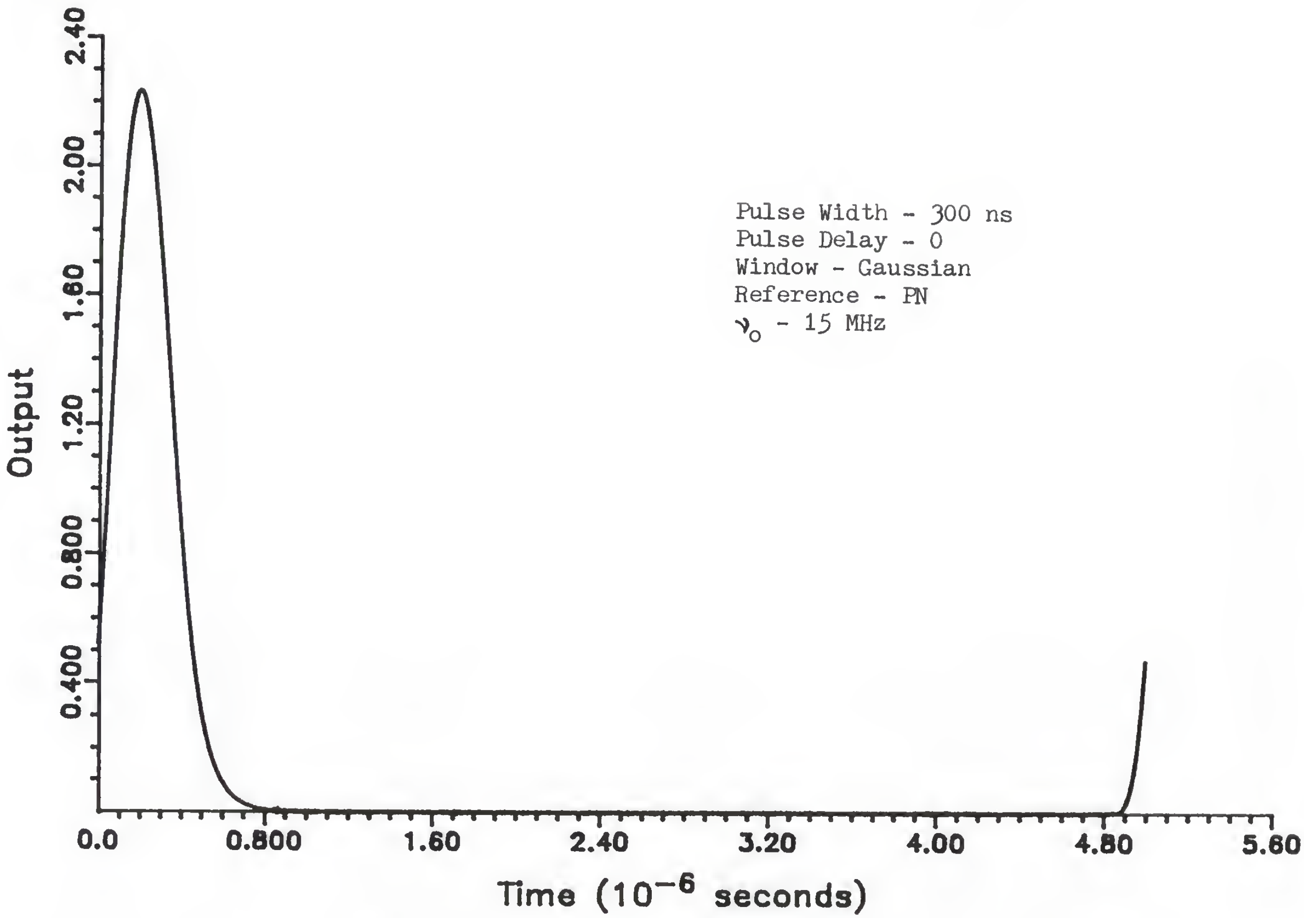

Figure 18. Output from Coherent Detector in Time Domain

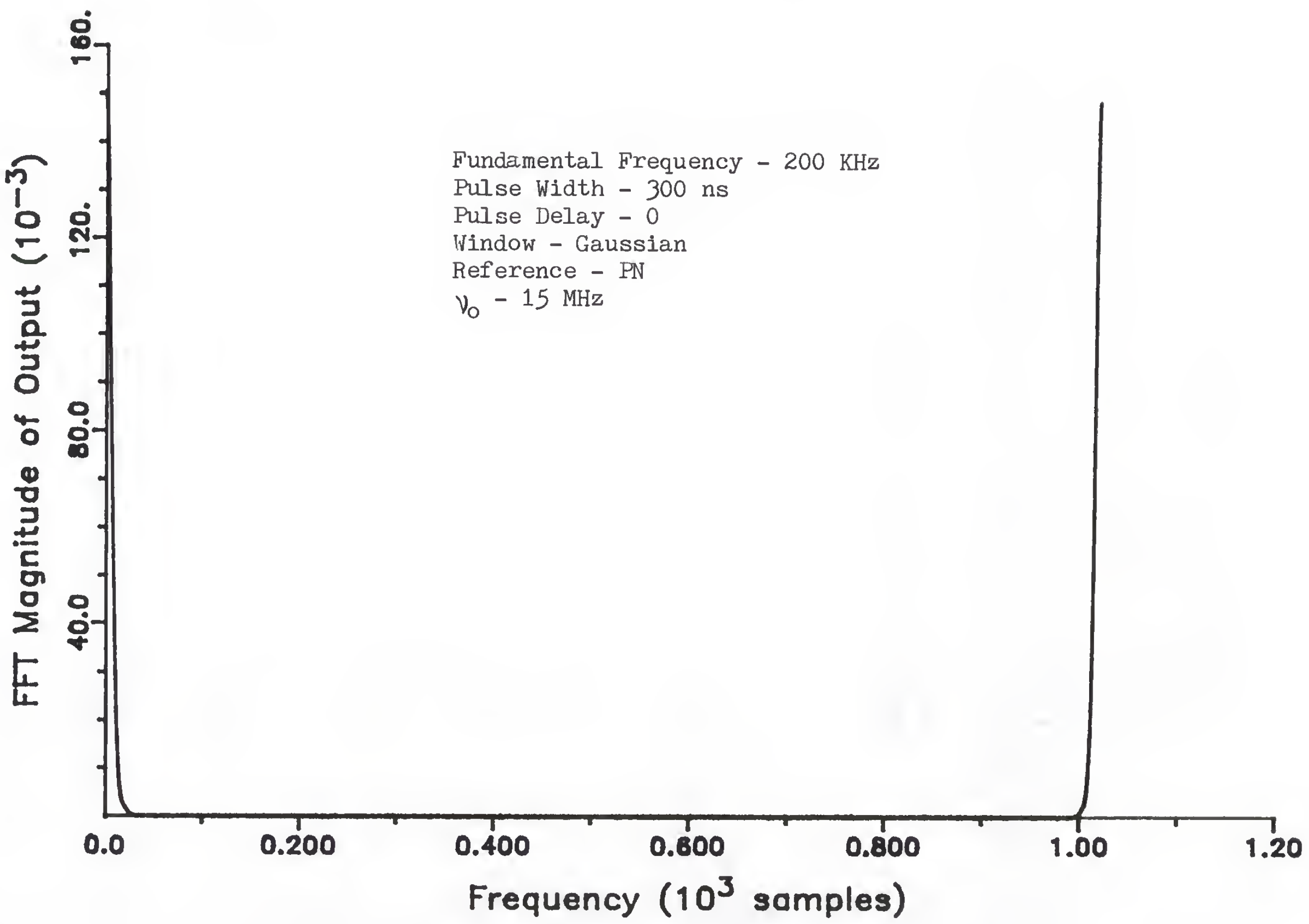

Figure 19. FFT of Output from Coherent Detector

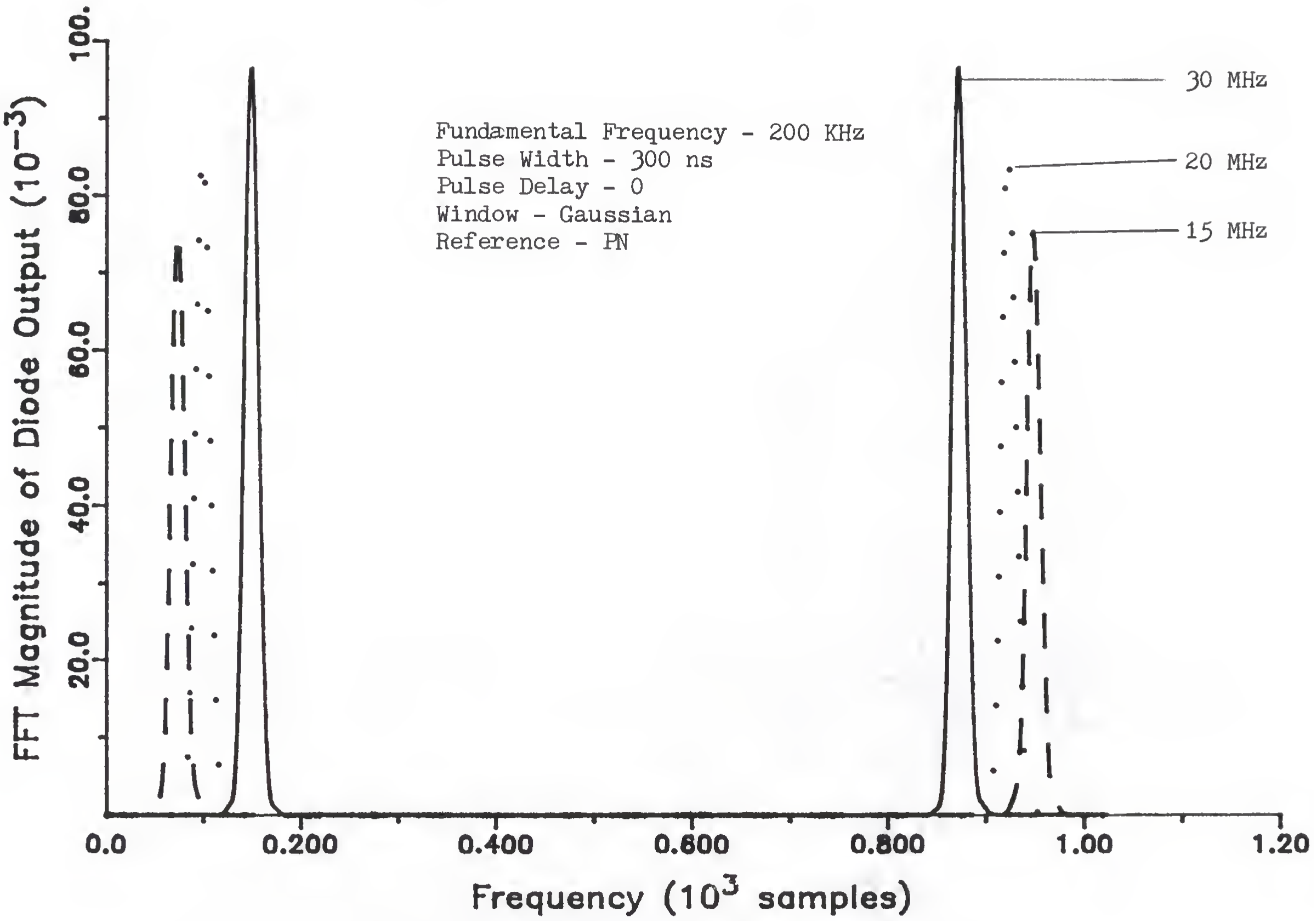

Figure 20. Effect of Changing ν_0

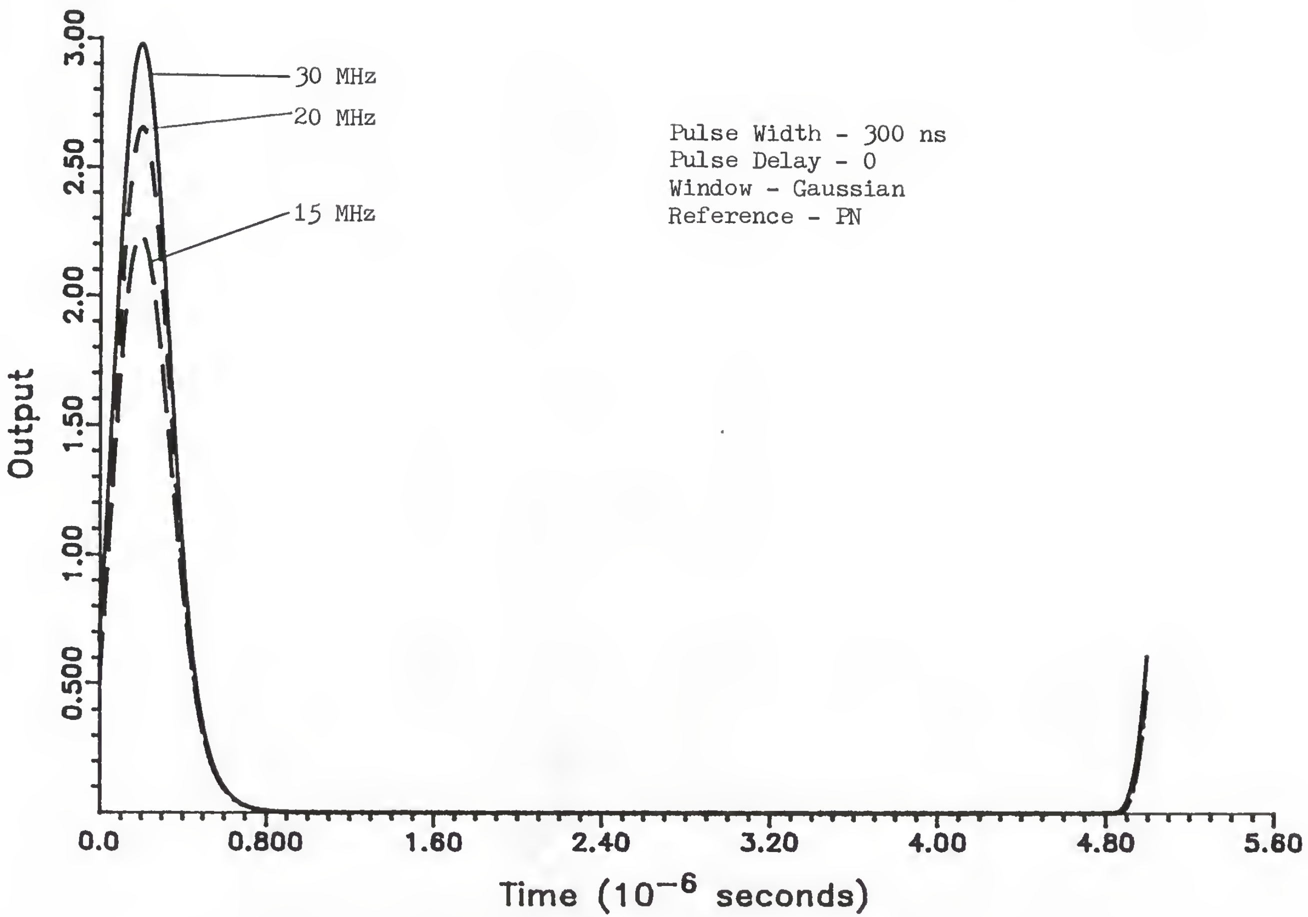

Figure 21. Effect of Changing ν_o on Output in Time Domain

Type of Reference Signal

As shown in Equation (26) the output of the IFAO spectrum analyzer is largely dependent on the reference signal. Figure 22 gives a diagram of Equation (26) which further explains the action of the analyzer. To get output from the analyzer the output from both the reference and the signal path must be non-zero simultaneously. A good reference should be continuously present and have a component to each that has a constant envelope. References must have frequency components with the frequency band of both the Bragg cell window and the diode aperture. Figure 23 shows the varied outputs from different reference signals. The sum of sinusoids provides the best output because it meets the above criterion very closely. The other reference signals produce less output amplitude because they contain frequencies that are not passed through and vary with time.

Type of Bragg Cell Window

Figure 16 shows that the rectangular window is somewhat broader than the Gaussian window, in terms of a spatial frequency band. This allows increased reference and signal energy to be passed with a resultant increase of output, as shown in Figure 24. Also notice that the output for the case of the rectangular window exhibits small variations, probably due to the (sin x)/x window function.

Carrier Frequency Offset

Figure 25 shows the effect of moving the carrier frequency of the input signal away from the center. To produce output the signal must be passed through both the Bragg cell and the diode aperture. These

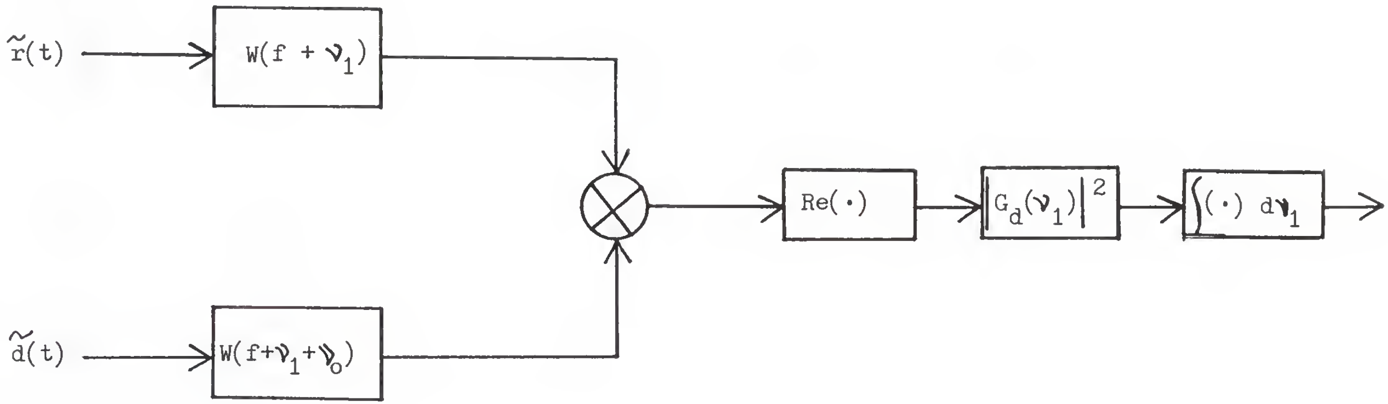

Figure 22. Simplified IFAO Model

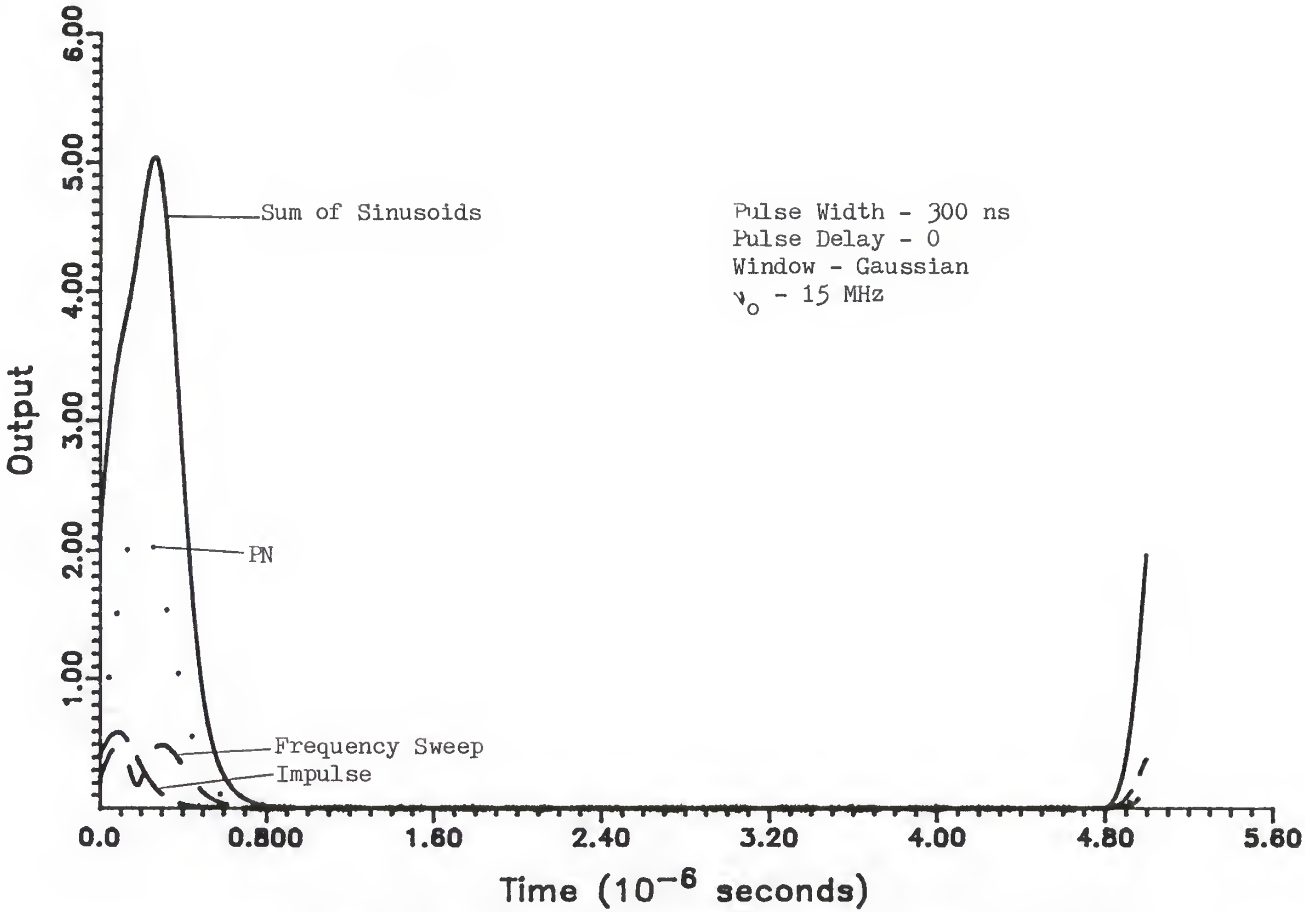

Figure 23. Effect of Different Reference Signals on Output

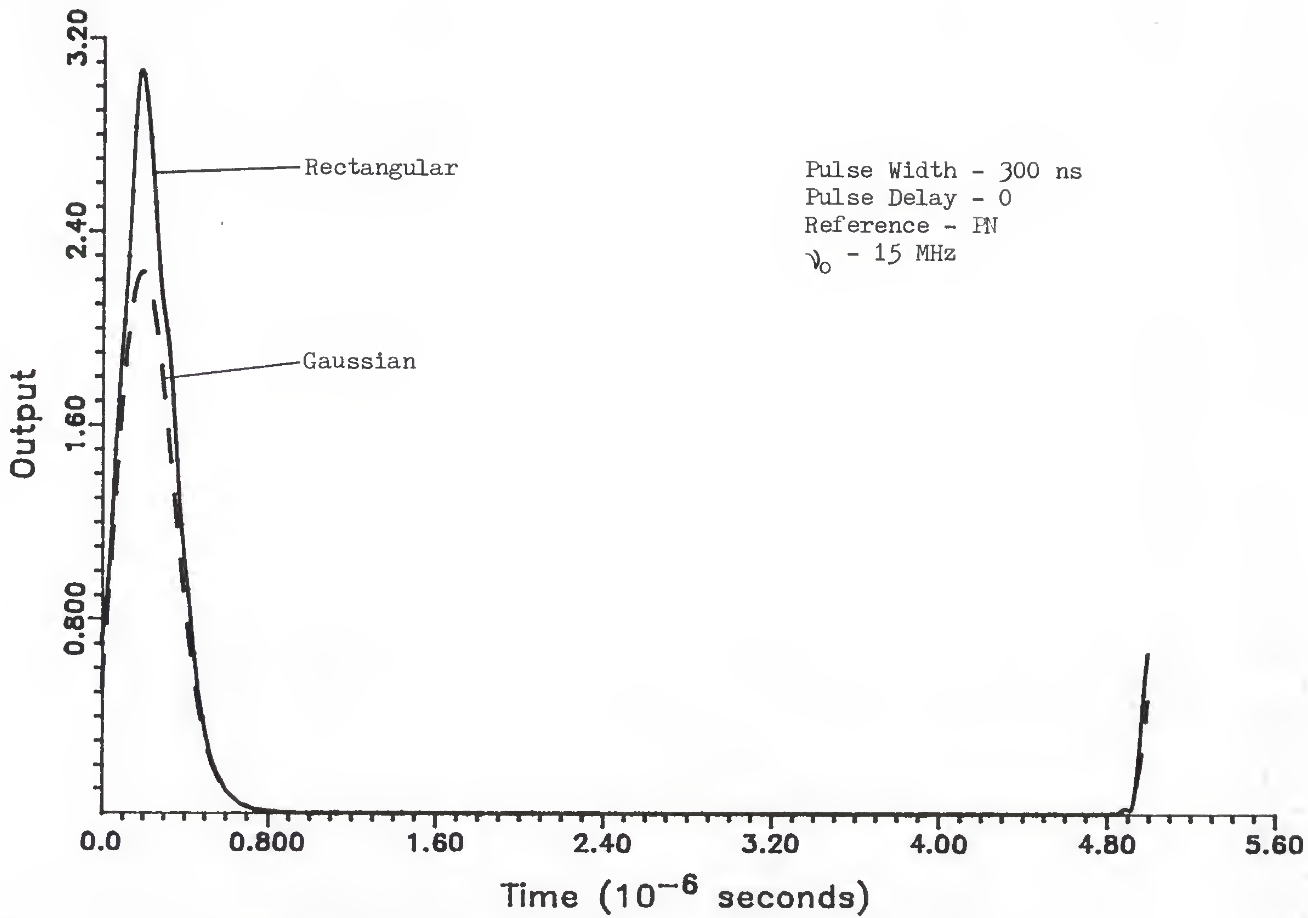

Figure 24. Effect of Bragg Cell Window on Output

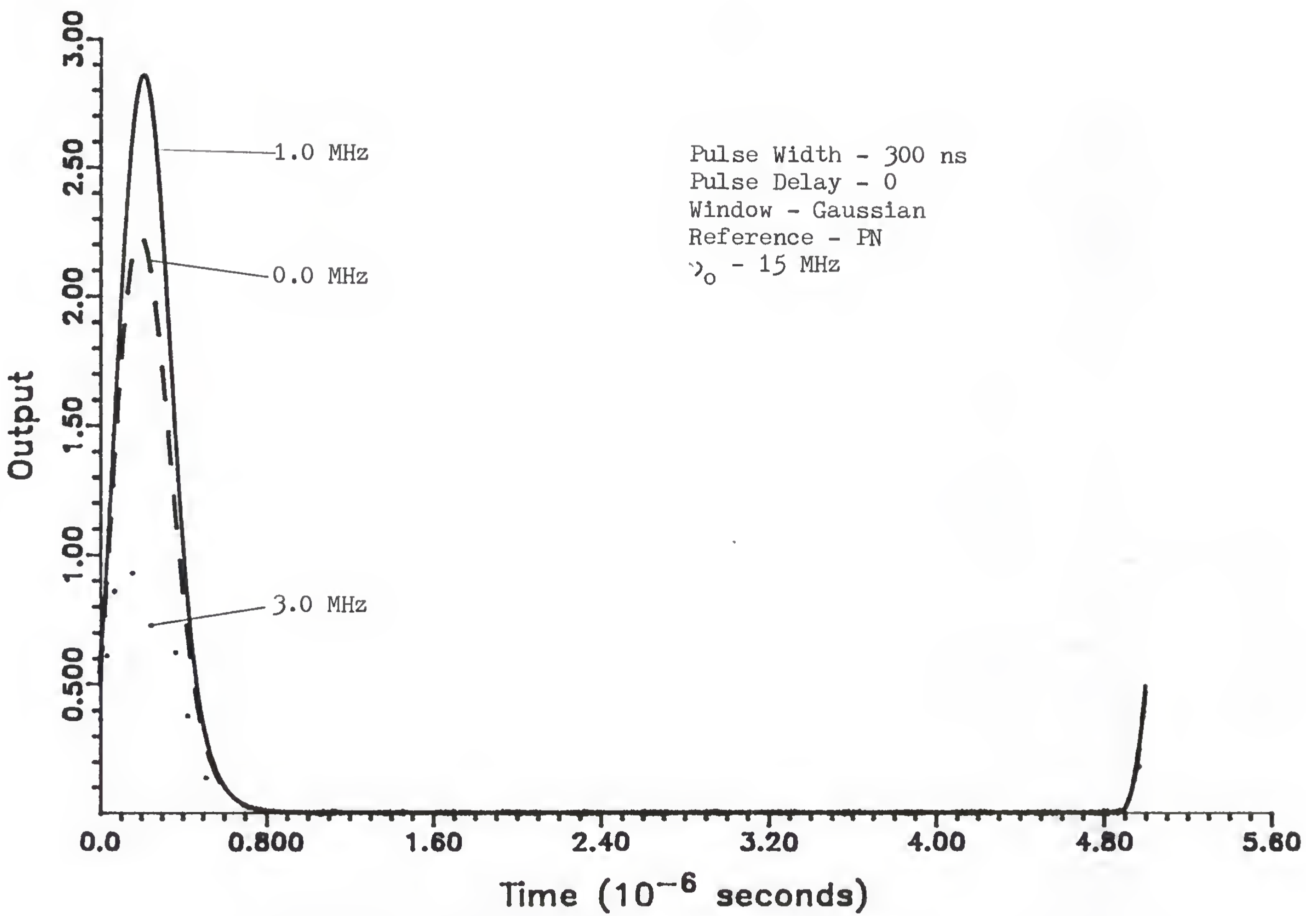

Figure 25. Effect of Carrier Offset on Output

devices have 3 dB equivalent bandwidths of about 4.0 MHz and 5.0 MHz respectively. Moving outside of these bandwidths should decrease output significantly. The 1.0 MHz offset has higher amplitude than no offset. This is probably due to differences in PN reference signals. The 3.0 MHz offset does show the expected decrease in amplitude and output shape. The results of the program may be used to predict the ability of the analyzer to resolve signals which are closely spaced in frequency.

Pulse Width

Pulses of width 100, 200, 300 and 400 ns were entered into the computer program. Figure 26 shows how differing pulse widths change the output. The program produced outputs that were proportional to the pulse width of the inputs. Because of the convolution operation of the Bragg cell, pulses of short length are "stretched". The time it takes a pulse to propagate across the Bragg cell lengthens the output to at least the length of the Bragg cell. Therefore shorter pulses are not as accurately reproduced as are longer pulses. This phenomena was also found in [4]. Note that the shorter pulses have lower amplitudes, due to less energy in the signal.

Pulse Delay

Delaying the pulses for different references provides further insight into the working of the IFAO spectrum analyzer and the importance of choosing a proper reference signal. The impulse reference signal is not considered for this case, since the output is near zero for any amount of delay.

For the PN reference, Figure 27 shows how variable the output is.

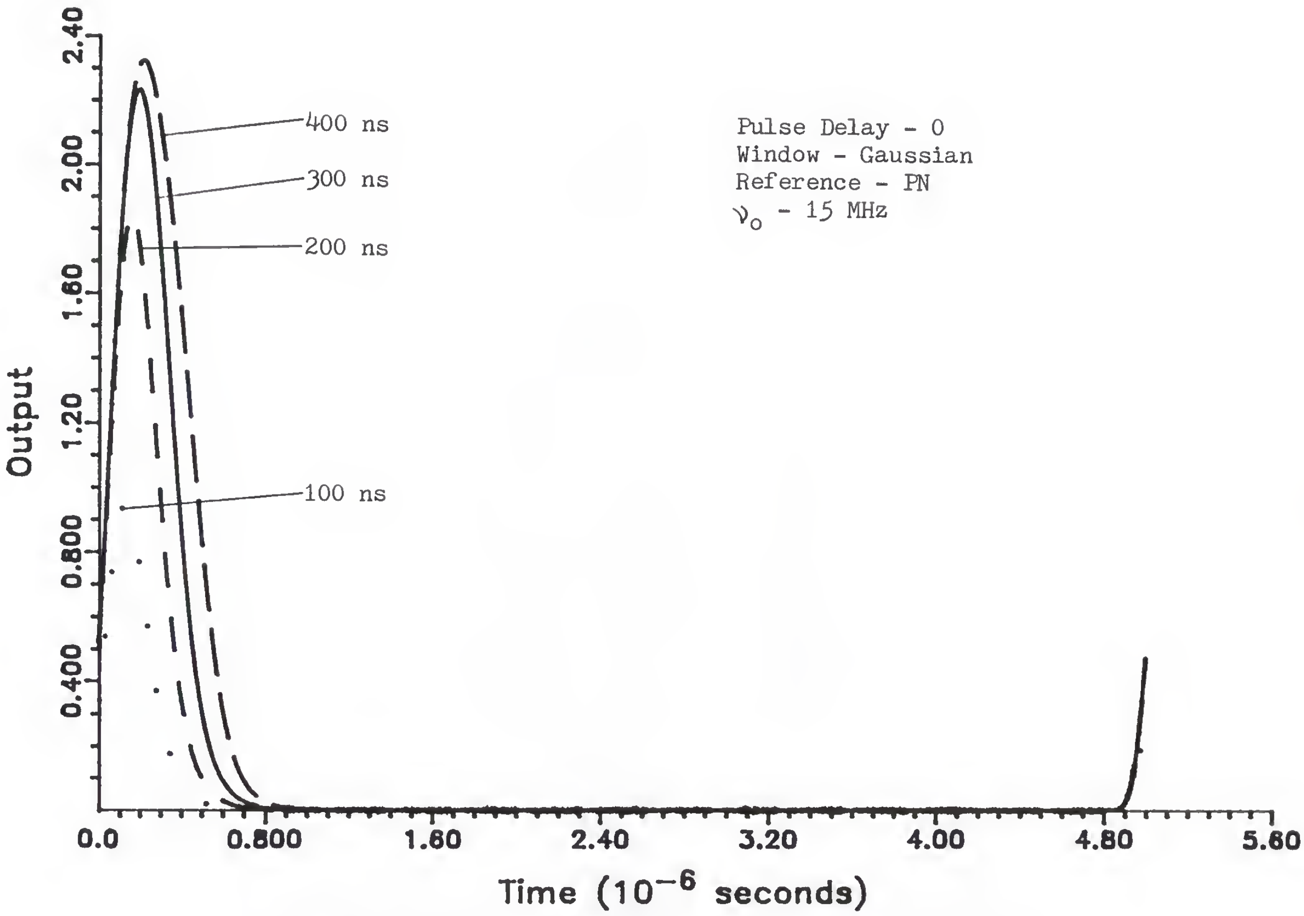

Figure 26. Effect of Different Pulse Widths on Output

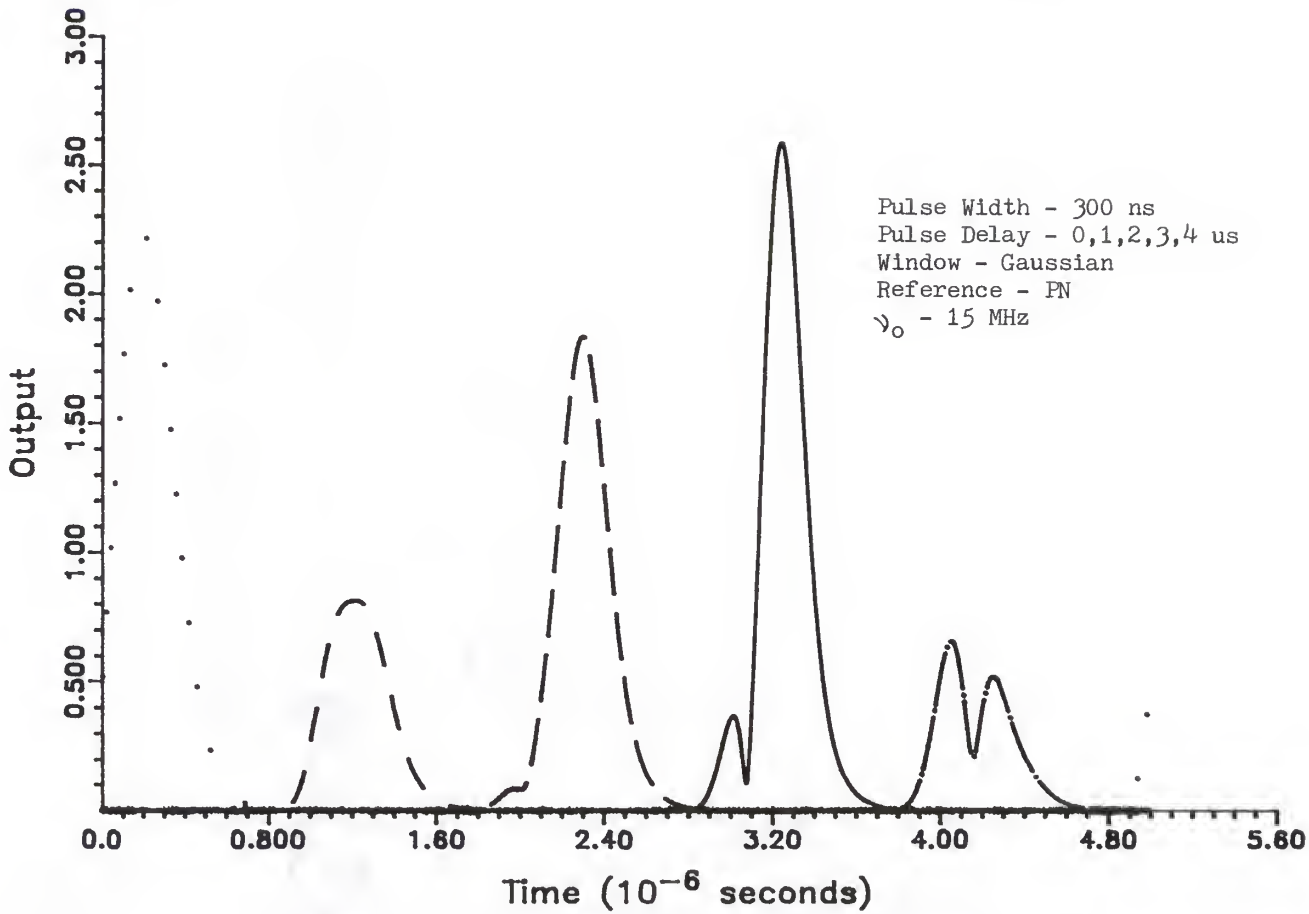

Figure 27. Pulse Delay for PN Reference Signal

The output of the analyzer is very dependent on the state of the PN sequence. At certain periods the PN reference does not provide the proper frequencies to pass through the Bragg cell and the diode aperture, thus the output is small.

The frequency sweep reference has a similar response. Figure 28 displays the pulse for several delay times. While the sweep is going through zero frequency, the output becomes small, because the low frequencies are not passed. When the frequency is near ν_0, the output amplitude is larger. This large dependence on the reference signal leads to the consideration of a sinusoidal reference signal. Such a signal would not exhibit the problems of either the PN or frequency sweep reference signals.

A sum of sinusoids, centered at ν_0 and spaced by the diode spacing, 4 MHz, was then used for reference. This type of reference would meet the requirements defined when observing Figure 22, the simplified IFAO model. The sum of sinusoids reference signal was relatively constant over time and was within the frequency bands needed. The output for the sum of sinusoids reference was very good for all delay times, see Figure 29. It appears that this type of reference signal is one of the best to consider for implementation because of the excellent results. However, as a practical matter, such signals are difficult to implement because of the large peak to RMS ratio they usually exhibit.

IV. CONCLUSIONS

A mathematical representation for the IFAO spectrum analyzer was developed, and a computer algorithm was produced. Then a computer program was implemented to emulate the analyzer using this algorithm.

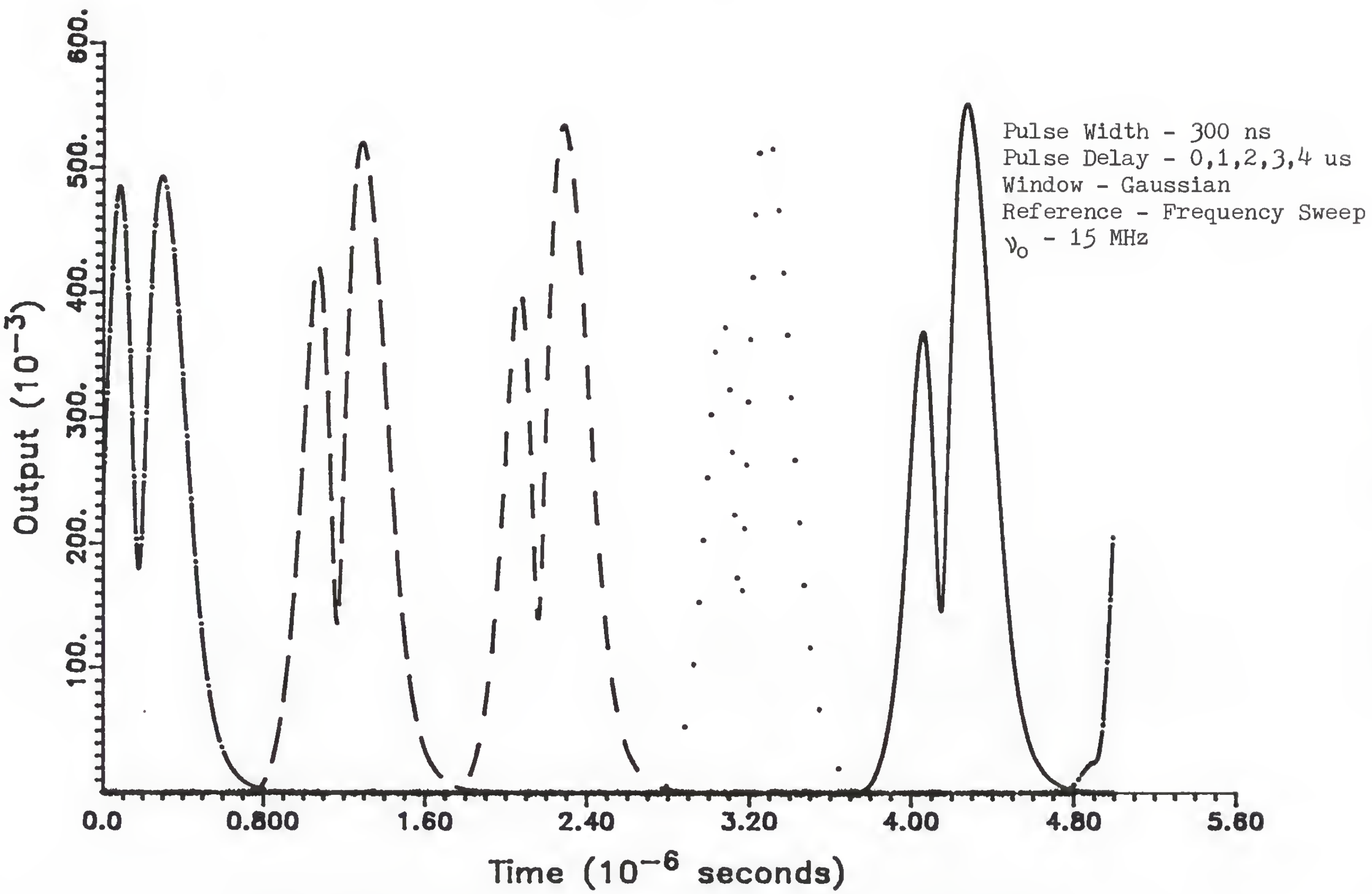

Figure 28. Pulse Delay for Frequency Sweep Reference Signal

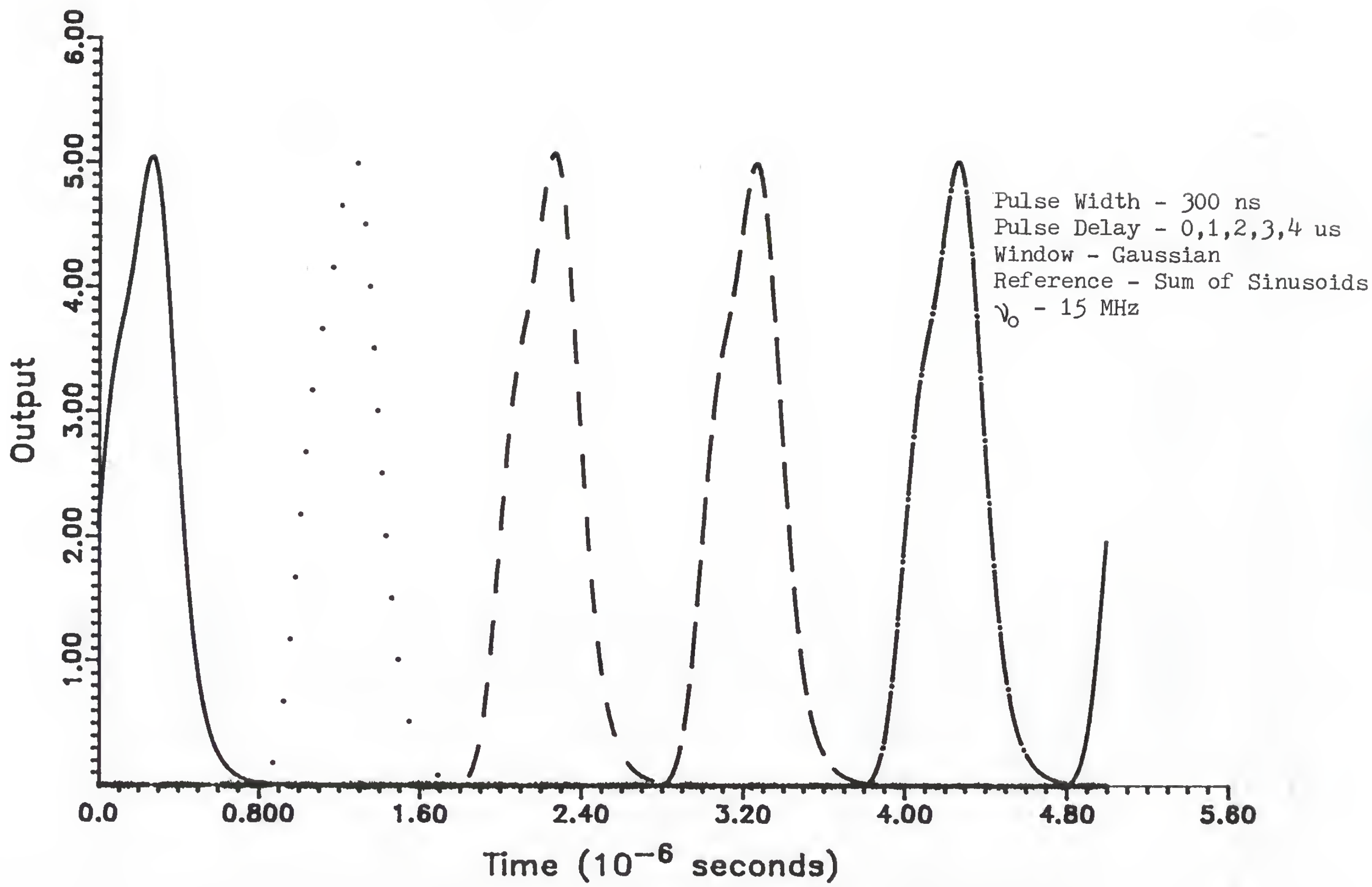

Figure 29. Pulse Delay for Sum of Sinusoids Reference Signal

Several test cases were then run through the program.

From the results it was found that the reference signal has a large influence on the performance of the IFAO. A reference signal that is centered around the frequency ν_0 and that provides a component to each diode that has a constant envelope was found to be the best. In practice a large diode array of possibly 100 or more diodes would be used. This type of implementation might make the sum of sinusoids reference signal more difficult to use. Therefore, the PN or the frequency sweep reference signal might be more appropriate.

Also, the signal and the reference must pass through both the Bragg cell window and the diode aperture to produce output. The properties of these system elements are a major factor in choosing a reference signal that gives good system performance.

Future work could be done in further evaluating different types of reference signals for maximal output. The effect of other parameters, such as Bragg cell window and diode aperture, in the IFAO could also be investigated. A further generalization of the mathematical model could be enlarged to include noise and provide a more extensive measure of system performance.

REFERENCES

[1] A. Vander Lugt, Interferometric Spectrum Analyzer, Applied Optics, Vol. 20, No. 16, August 15, 1981.

[2] M. King, W. R. Bennett, L. B. Lambert and M. Arm, Real Time Electro–optical Signal Processors with Coherent Detection, Applied Optics, Vol. 6, No. 8, August, 1967.

[3] L. M. Ralston, A. M. Bardos, "Wideband, Interferometric Spectrum Analyzer Improvement", AFWAL–TR–84–1029, Harris Corp., Melbourne, FL 32902, Submitted to AFWAL/AADO–2, Wright–Patterson AFB, Ohio 45433, May 7, 1984.

[4] B. K. Harms, D. R. Hummels, "An Analysis and Comparison of the Channelized, Acoustooptic, and Frequency Compressive Intercept Receivers", Final Report, Volume 1, Kansas State University Engineering Experiment Station, Project 2851, Submitted to Motorola Inc., Government Electronics Group, June 3, 1985.

Appendix

Computer Programs

```
/*******************************************************************

        Source File Name:       ao_recvr.c

        Calling Sequence:       main()

        Usage:  This program models an acoustoptic interferomet-
                ric receiver.  This implementation uses a 1-pole
                bandpass filter in the lowpass equivalent and
                phase invarient coherent detection.

        Parameters:     None.

        Return:         None

        Author: Alan L. Ferguson

        Date:   5/27/88

*******************************************************************/

#include "amath.h"              /*  Complex math file          */
#include "ao_recvr.h"           /*  Special definitions        */
#include <math.h>
#include <stdio.h>

/*  Declare arrays (complex and doubles)                           */

    COMPLEX     output1[N_POINTS], output2[N_POINTS],
                reference[N_POINTS+6], result[N_POINTS],
                resultc[N_POINTS], results[N_POINTS],
                r_row[N_POINTS], signal[N_POINTS],
                s_row[N_POINTS], temp[N_POINTS+6];

    double      plot[N_POINTS], wind_diode[2*F_POINTS];

main()
{
    int         error, four, i, j, k, nine, out,
                shift_reg, sum, tap, type, wtype;

    long int    differ, f_shift;

    double      amplitude, delta_f, delta_t, frequency,
                nu, nu_0, r_time, p_delay, p_width,
                sample_f, s_time, t, wind_f;

    char        df[NAME_LEN], dt[NAME_LEN],
                name[NAME_LEN],
                of[NAME_LEN], ot[NAME_LEN],
                rf[NAME_LEN], rt[NAME_LEN];

    COMPLEX     window(), cfilter();
```

```
    FILE        *data, *sig, *ref, *wind;

/*  Initialize result array and others                                    */
    for(i = 0; i < N_POINTS; i++)
        result[i] = cmplx(ZERO, ZERO);

    nine       = 0x0080; /* Output at bit 4                               */
    four       = 0x1000; /* Tap at bit 4                                  */
    shift_reg = 0x1010; /* Seed for random number generator              */

/*----------------------------------------------------------------------*/
/*  Enter parameters about signal and system                              */
/*----------------------------------------------------------------------*/
/*    puts("Data file?");
    scanf("%s", name);

    data = fopen(name, "r");

    fscanf(data, "%lf", &s_time);


    fscanf(data, "%lf", &p_width);

    fscanf(data, "%lf", &r_time);

    fscanf(data, "%lf", &p_delay);

    fscanf(data, "%lf", &amplitude);

    fscanf(data, "%lf", &delta_f);

    fscanf(data, "%lf", &nu_0);

    fscanf(data, "%d", &wtype);

    fscanf(data, "%d", &type);

    fscanf(data, "%s", dt);

    fscanf(data, "%s", df);

    fscanf(data, "%s", ot);

    fscanf(data, "%s", of);

    fscanf(data, "%s", rt);

    fscanf(data, "%s", rf);


    fclose(data);*/
```

```
/*  Input parameters by hand                                          */
    puts("Enter sample length (time)");
    scanf("%lf", &s_time);

    puts("Enter pulse width");
    scanf("%lf", &p_width);

    puts("Enter rise time");
    scanf("%lf", &r_time);

    puts("Enter pulse delay");
    scanf("%lf", &p_delay);

    puts("Enter amplitude");
    scanf("%lf", &amplitude);

    puts("Enter frequency difference");
    scanf("%lf", &delta_f);

    puts("Enter Nu 0");
    scanf("%lf", &nu_0);

    puts("Enter type of Bragg Cell aperature window in time");
    puts("(0 for Gaussian, 1 for Rectangular)");
    scanf("%d", &wtype);

    puts("Enter 0 for impulse 1 for PN sequence, 2 for sweep");
    scanf("%d", &type);

    puts("Reference file name (time)");
    scanf("%s", rt);

    puts("Reference file name (frequency)");
    scanf("%s", rf);

    puts("Diode output file name (time)");
    scanf("%s", dt);

    puts("Diode output file name (frequency)");
    scanf("%s", df);

    puts("Output file name (time)");
    scanf("%s", ot);

    puts("Output file name (frequency)");
    scanf("%s", of);

/*------------------------------------------------------------------*/
/*  Set up the signal sequence                                      */
/*------------------------------------------------------------------*/
    i = 0;
    delta_t = s_time / (double)N_POINTS;
    sample_f = ONE / s_time;
```

```
    printf("%lf is sample freq\n", sample_f);

/*  Set up initial zero                                         */
    while(i * delta_t < p_delay)
        {
        signal[i] = cmplx(ZERO, ZERO);
        i++;
        }

/*  Set up rising edge                                          */
    while(i * delta_t < r_time + p_delay)
        {
        signal[i] = cmplx((amplitude / r_time)*
            (i * delta_t - p_delay), ZERO);
        i++;
        }

/*  Set up plateau                                              */
    while(i * delta_t < p_width + p_delay)
        {
        signal[i] = cmplx(amplitude, ZERO);
        i++;
        }

/*  Set up falling edge                                         */
    while( i * delta_t < r_time + p_delay + p_width)
        {
        signal[i] = cmplx((p_width + r_time + p_delay - i *
            delta_t) * (amplitude / r_time), ZERO);
        i++;
        }

/*  Set up trailing zero                                        */
    for(j = i; j < N_POINTS; j++)
        signal[j] = cmplx(ZERO, ZERO);

/*  Add frequency shift from center                             */
    for(i = 0; i < N_POINTS; i++)
        signal[i] = cmult(signal[i], cexpon(2.0*PI*i*delta_t*
            delta_f));

/*  Write signal in time domain to disk                         */
/*    puts("Signal File name (time)");
    scanf("%s", name);
    sig = fopen(name, "w");
    fprintf(sig, "%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        fprintf(sig, "%d %lf \n", i, signal[i].re);
    fclose(sig);                                                */
```

```
/*****************************************************************/
/*  Transform the signal to the frequency domain                 */
/*****************************************************************/
    error = cfft(signal, N_POINTS, FORWARD);
    puts("FFT done");

/*  Write signal in frequency domain to disk                     */
/*    puts("Signal file name (frequency)");
    scanf("%s", name);
    sig = fopen(name, "w");
    fprintf(sig, "%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        {
        plot[i] = cmag(signal[i]);
        fprintf(sig, "%d %lf\n", i, plot[i]);
        }

    fclose(sig);                                                 */


/*****************************************************************/
/*  Create reference signal in time domain                       */
/*****************************************************************/
    switch(type)
        {
/*      It is a impulse at zero for this case                    */
        case 0:
            reference[0] = cmplx(5.0, ZERO);
            for(i = 1; i < N_POINTS; i++)
                reference[i] = cmplx(ZERO,ZERO);
            break;

/*      Pseudo-random noise generator                            */
        case 1:
            for(i = 0; i < (int)N_POINTS/5; i++)
                {
/*              Check for output bit set                         */
                if((shift_reg & nine) != ZERO)
                    out = 0x8000;
                else
                    out = 0x0000;

/*              Check for tap bit set                            */
                if((shift_reg & four) != ZERO)
                    tap = 0x8000;
                else
                    tap = 0x0000;

/*              Shift register rolling in sum of tap and output*/
                sum = tap ^ out;
                shift_reg = shift_reg >> 1;
                shift_reg = shift_reg & 0x7fff;
                shift_reg = shift_reg | sum;
```

```
/*              Check output                                        */
                if(out != ZERO)
                    for(j = 0; j < 5; j++)
                        reference[i*5 + j] = cmplx(ONE, ZERO);
                else
                    for(j = 0; j < 5; j++)
                        reference[i*5 + j] = cmplx(-ONE, ZERO);
                }

/*          Multiply by exp(j2PInu_ot)                              */
            for(i = 0; i < N_POINTS; i++)
                {
                t = (double)i * delta_t;
                reference[i] = cmult(reference[i],
                    cexpon(2.0*PI*t*nu_0));
                }

            break;

/*          This is the case for the frequency sweep (40 MHz)  */
        case 2:
            for(j = 0; j < 20; j++)
                {
                for(i = 0; i < (int)N_POINTS/20; i++)
                    {
                    t = (double) i * delta_t;
                    reference[(int)j*(N_POINTS/20) + i] = cexpon(
                        20.0*2.0*PI*t*t*40.0e6*
                        sample_f - 2.0*PI*40.0e6*t);
                    }
                }

/*          Rotate the frequency in time                            */
/*            for(i = 0; i < N_POINTS/2; i++)
                temp[i] = reference[i + N_POINTS/2];
            for(i = N_POINTS/2 ; i < N_POINTS; i++)
                temp[i] = reference[i - N_POINTS/2];

            for(i = 0; i < N_POINTS; i++)
                reference[i] = temp[i];*/

            break;

/*          Sine wave case                                          */
        case 3:
            for(i = 0; i < N_POINTS; i++)
                {
                t = delta_t * i;
                reference[i] = cexpon(2.0*PI*nu_0*t);
                }

            break;
```

```
/*          Sum of sinusoids                                              */
        case 4:
            for(j = -2; j < 3; j++)
                {
                for(i = 0; i < N_POINTS; i++)
                    {
                    t = delta_t * i;
                    reference[i] = cadd(reference[i],
                        cexpon(2.0 * PI *(nu_0 + j * 4.0e6) * t));
                    }
                }

            break;
            }

puts("done");

/*  Write reference in time domain to disk                               */
/*    ref = fopen(rt, "w");
    fprintf(ref, "%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        fprintf(ref, "%d %lf\n", i, reference[i].re);
    fclose(ref);*/


/*  Transform to frequency domain                                        */
    error = cfft(reference, N_POINTS, FORWARD);
    puts("After cfft for ref");

/*  Write reference in frequency domain to disk                          */
/*    ref = fopen(rf, "w");
    fprintf(ref, "%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        {
        plot[i] = cmag(reference[i]);
        fprintf(ref, "%d %lf\n", i, plot[i]);
        }

    fclose(ref);*/


    puts("Reference done!");

/*************************************************************************/
/*  Create Photodiode frequency window                                   */
/*************************************************************************/
    i = 0;
    nu = 3.75e6 / (double)F_POINTS;
```

```
/*  Create plateau                                                  */
    while((double)i*nu < 1.25e6)
        {
        wind_diode[F_POINTS+i] = ONE;
        wind_diode[F_POINTS-i] = ONE;
        i++;
        }

/*  Create sloped sides                                             */
    while((double)i*nu < 3.75e6)
        {
        wind_diode[F_POINTS+i] = -4e-7 * i*nu + 1.5;
        wind_diode[F_POINTS-i] = -4e-7 * i*nu + 1.5;
        i++;
        }

/*  Output to file the diode aperature window                       */
/*    ref = fopen("diode.dat", "w");
    fprintf(ref, "%d \n", 2*F_POINTS);
    for(i = 0; i < 2*F_POINTS; i++)
        fprintf(ref, "%d %lf\n", i, wind_diode[i]);
    fclose(ref);                                                    */

/*################################################################*/
/******************************************************************/
/*  Step through signal matrix a row at a time                      */
/******************************************************************/
/*################################################################*/
    for(j = -F_POINTS+1; j < F_POINTS; j++)
        {
        printf("Iteration %3d\r", j+F_POINTS);

/******************************************************************/
/*      Create signal row                                           */
/******************************************************************/
        s_row[0]          = cmult(signal[0], window(wtype,nu*j));
        s_row[N_POINTS/2] = cmult(signal[N_POINTS/2],
            window(wtype, sample_f*N_POINTS/2 + nu*j));
        for(k = 1; k < N_POINTS/2; k++)
            {
            s_row[k] = cmult(signal[k], window(wtype, sample_f*k
                +nu*j));
            s_row[N_POINTS-k] = cmult(signal[N_POINTS-k],
                window(wtype, -sample_f*k + nu*j));
            }

/*      Inverse transform the row of the signal matrix              */
        error = cfft(s_row, N_POINTS, INVERSE);
```

```
/*****************************************************************/
/*      Create reference  row                                    */
/*****************************************************************/
        r_row[0] = cmult(reference[0], window(wtype, nu*j +
            nu_0));
        r_row[N_POINTS/2] = cmult(reference[N_POINTS/2],
            window(wtype, sample_f*N_POINTS/2 + nu*j + nu_0));
        for(k = 1; k < N_POINTS/2; k++)
            {
            r_row[k] = cmult(reference[k], window(wtype,
                sample_f*k + nu*j + nu_0));
            r_row[N_POINTS-k] = cmult(reference[N_POINTS-k],
                window(wtype, -sample_f*k + nu*j + nu_0));
            }

/*      Inverse transform the row of the reference matrix        */
        error = cfft(r_row, N_POINTS, INVERSE);

/*****************************************************************/
/*      Multiply rows element by element taking real part only */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            {
            s_row[i] = cmult(s_row[i], cconj(r_row[i]));
            s_row[i].im = ZERO;
            }

/*****************************************************************/
/*      Multiply rows by diode aperature sequence                */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            s_row[i].re = s_row[i].re * wind_diode[F_POINTS+j];

/*****************************************************************/
/*      Form resulting array                                     */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            result[i].re = result[i].re + s_row[i].re;

/*****************************************************************/
/*      Create signal row (second term)                          */
/*****************************************************************/
        s_row[0]          = cmult(cconj(signal[0]), window(wtype,
            -nu*j));
        s_row[N_POINTS/2] = cmult(cconj(signal[N_POINTS/2]),
            window(wtype, sample_f*N_POINTS/2 - nu*j));
        for(k = 1; k < N_POINTS/2; k++)
            {
            s_row[k] = cmult(cconj(signal[N_POINTS-k]),
                window(wtype, sample_f*k - nu*j));
            s_row[N_POINTS-k] = cmult(cconj(signal[k]),
                window(wtype, -sample_f*k - nu*j));
            }
```

```
/*      Inverse transform the row of the signal matrix            */
        error = cfft(s_row, N_POINTS, INVERSE);

/*****************************************************************/
/*      Create reference  row (second term)                       */
/*****************************************************************/
        r_row[0] = cmult(cconj(reference[0]),
            window(wtype, -nu*j + nu_0));
        r_row[N_POINTS/2] = cmult(cconj(reference[N_POINTS/2]),
            window(wtype, sample_f*N_POINTS/2 - nu*j + nu_0));
        for(k = 1; k < N_POINTS/2; k++)
            {
            r_row[k] = cmult(cconj(reference[N_POINTS-k]),
                window(wtype, sample_f*k - nu*j + nu_0));
            r_row[N_POINTS-k] = cmult(cconj(reference[k]),
                window(wtype, -sample_f*k - nu*j + nu_0));
            }

/*      Inverse transform the row of the reference matrix         */
        error = cfft(r_row, N_POINTS, INVERSE);

/*****************************************************************/
/*      Multiply rows element by element taking real part only */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            {
            s_row[i] = cmult(s_row[i], cconj(r_row[i]));
            s_row[i].im = ZERO;
            }

/*****************************************************************/
/*      Multiply rows by diode aperature sequence                 */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            s_row[i].re = s_row[i].re * wind_diode[F_POINTS+j];

/*****************************************************************/
/*      Form resulting array                                      */
/*****************************************************************/
        for(i = 0; i < N_POINTS; i++)
            result[i].re = result[i].re + s_row[i].re;
        }

/*  Write diode output in time domain to disk                     */
/*    ref = fopen(dt, "w");
    fprintf(ref,"%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        fprintf(ref, "%d %e\n", i, result[i].re);
    fclose(ref);*/
```

```
/*****************************************************************/
/*  Transform                                                    */
/*****************************************************************/
    error = cfft(result, N_POINTS, FORWARD);

/*  Write diode output in frequency domain to disk               */
/*    ref = fopen(df, "w");
    fprintf(ref,"%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        {
        plot[i] = cmag(result[i]);
        fprintf(ref, "%d %e\n", i, plot[i]);
        }

    fclose(ref);*/

/*****************************************************************/
/*  Bandpass signal to retrieve signal w/ 1-Pole filter          */
/*****************************************************************/
    puts("Bandpass filter of signal");

    result[0] = cmult(cfilter(0.0, 1, 2.0E6 ), result[0]);
    result[N_POINTS/2] = cmult(cfilter((N_POINTS/2)*sample_f-
        nu_0, 1, 2.0E6), result[N_POINTS/2]);

    for(i = 1; i < N_POINTS/2; i++)
        {
        result[i] = cmult(cfilter((i*sample_f)-nu_0, 1, 2.0E6),
            result[i]);
        result[N_POINTS-i] = cmult(cfilter((-i*sample_f)+nu_0, 1,
            2.0E6), result[N_POINTS-i]);
        }

/*  Output to file the filtered response                         */
/*    ref = fopen("filter", "w");

    for(i = 0; i < N_POINTS; i++)
        fprintf(ref, "%lf\n", cmag(result[i]));

    fclose(ref);*/

/*****************************************************************/
/*  Detect signal envelope using coherent detection              */
/*****************************************************************/
    puts("Detecting output envelope");
    f_shift = (long) (nu_0 / sample_f);
    differ  = (long) (2 * (N_POINTS/2 - f_shift));

/*  Phi = 0 (cosine)                                             */
/*  Case of X(f + fc)                                            */
    for(i = 0; i < differ+f_shift; i++)
        output1[i] = result[i+f_shift];
```

```
    for(i = differ+f_shift; i < N_POINTS; i++)
        output1[i] = result[i-(f_shift+differ)];

/*  Case of X(f - fc)                                          */
    for(i = 0; i < f_shift; i++)
        output2[i] = result[i+f_shift+differ];

    for(i = f_shift; i < N_POINTS; i++)
        output2[i] = result[i-f_shift];

    for(i = 0; i < N_POINTS; i++)
        resultc[i] = cadd(output1[i], output2[i]);

/*  Implement brick wall lowpass filter                        */
    for(i = f_shift; i < N_POINTS-f_shift; i++)
        resultc[i] = cmplx(ZERO, ZERO);

/*  Phi = 90 degrees (sine)                                    */
/*  Case of X(f + fc)                                          */
    for(i = 0; i < differ+f_shift; i++)
        output1[i] = cmult(cexpon(-PI/2.0), result[i+f_shift]);

    for(i = differ+f_shift; i < N_POINTS; i++)
        output1[i] = cmult(cexpon(-PI/2.0), result[i-(f_shift+
            differ)]);

/*  Case of X(f - fc)                                          */
    for(i = 0; i < f_shift; i++)
        output2[i] = cmult(cexpon(PI/2.0), result[i+f_shift+
            differ]);

    for(i = f_shift; i < N_POINTS; i++)
        output2[i] = cmult(cexpon(PI/2.0), result[i-f_shift]);

    for(i = 0; i < N_POINTS; i++)
        results[i] = cadd(output1[i], output2[i]);

/*  Implement brick wall lowpass filter                        */
    for(i =f_shift; i < N_POINTS-f_shift; i++)
        results[i] = cmplx(ZERO, ZERO);


/*************************************************************/
/*  Transform back to time domain                            */
/*************************************************************/
    error = cfft(resultc, N_POINTS, INVERSE);
    error = cfft(results, N_POINTS, INVERSE);
```

```
/*  Put sine and cosine detected arrays together for output      */
    for(i = 0; i < N_POINTS; i++)
        {
        result[i].re = sqrt(results[i].re*results[i].re +
            resultc[i].re*resultc[i].re);
        result[i].im = ZERO;
        }

/*  Output file in time domain to disk                            */
    ref = fopen(ot, "w");
    fprintf(ref,"%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        {
        fprintf(ref, "%d %e\n", i, result[i].re);
        }
    fclose(ref);

/*  Switch back to frequency domain                               */
    error = cfft(result, N_POINTS, FORWARD);

/*  Output frequency domain of output signal                      */
/*    ref = fopen(of, "w");
    fprintf(ref,"%d \n", N_POINTS);
    for(i = 0; i < N_POINTS; i++)
        {
        plot[i] = cmag(result[i]);
        fprintf(ref, "%d %e\n", i, plot[i]);
        }
    fclose(ref);*/

}
```

```
/*****************************************************************

        Source File Name:       ao_recvr.h

        Calling Sequence:       #include ao_recvr.h

        Usage:  Include file for ao_recvr.c.  Defines number of
                points used for sequences and iterations.

        Parameters:     None.

        Return:         None.

        Author: Alan L. Ferguson

        Date:   5/27/1988

*****************************************************************/

#define N_POINTS        1024    /*  Points in sequence          */
#define F_POINTS        10      /*  Frequencies for integration*/
#define NAME_LEN        30      /*  Length of file names        */
```

```c
/*****************************************************************

        Source File Name:       amath.h

        Calling Sequence:       #include "amath.h"

        Usage:  Complex structure definition and other math.

        Parameters:             None.

        Author: Alan L. Ferguson

        Date:   1/4/87

*****************************************************************/

/*Define complex data structure                                  */
typedef struct complex
{
    double      re,im;
} COMPLEX;

/*  Define useful constants                                      */
#define ZERO    0.0
#define ONE     1.0
#define TWO     2.0
#define PI      3.141592654

/*  Define complex math routines                                 */
COMPLEX cadd();
COMPLEX cconj();
COMPLEX cdiv();
COMPLEX cexpon();
double  cmag();
double  cmagsq();
COMPLEX cmplx();
COMPLEX cmult();
COMPLEX cneg();
double  cphase();
double  cphased();
COMPLEX csqrt();
COMPLEX csub();
double  creal();
double  cimag();
int     cfft();
COMPLEX cpow();
double  sinc();
double  ccabs();

/*  Define forward and inverse for FFT routine                   */
#define FORWARD         1
#define INVERSE         -1
```

```
/***************************************************************

        Source File Name:       cadd.c

        Calling Sequence:       cadd(z1,z2)

        Usage:  This routine adds two COMPLEX numbers.

        Parameters:     z1,z2   COMPLEX
                        The numbers to be added.

        Return:         COMPLEX
                        The resulting complex addition.

        Author: Alan L. Ferguson

        Date:   11/16/87

***************************************************************/
#include "amath.h"

COMPLEX cadd(z1,z2)
    COMPLEX z1,z2;
{
    COMPLEX result;

    result.re = z1.re + z2.re;
    result.im = z1.im + z2.im;

    return(result);
}
```

```
/************************************************************

        Source File Name:       ccabs.c

        Calling Sequence:       ccabs(z)

        Usage:  This routine calculates the absolute value
                of a COMPLEX number.

        Parameters:         z       COMPLEX
                            The number to find the absolute
                            value of.

                            cabs    double
                            The resulting absolute value.

        Author: Alan L. Ferguson

        Date:   7/28/87

************************************************************/
#include "amath.h"
#include <math.h>

double ccabs(z)
    COMPLEX z;
{
    double result;

    result = sqrt(z.re*z.re + z.im*z.im);
    return(result);
}
```

```
/****************************************************************

        Source File Name:       cconj.c

        Calling Sequence:       cconj(z)

        Usage:  This routine returns the conjugate of  z.

        Parameters:     z       COMPLEX
                        The COMPLEX variable of interest.

        Return:         COMPLEX
                        The conjugate of the argument.

        Author: Alan L. Ferguson

        Date:   11/16/87

****************************************************************/
#include "amath.h"

COMPLEX cconj(z)
    COMPLEX z;
{
    COMPLEX result;
    result.re = z.re;
    result.im = -(z.im);
    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cdiv.c

        Calling Sequence:       cdiv(z1,z2)

        Usage:  This routine performs a COMPLEX division
                on z1 and z2.

        Parameters:     z1,z2   COMPLEX
                        The COMPLEX numbers to be multiplied

        Return:         COMPLEX
                        The resulting division.

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/

#include "amath.h"

COMPLEX cdiv(z1,z2)
    COMPLEX z1,z2;
{
    COMPLEX result;
    double temp;

    COMPLEX cmult(),cconj();
    double creal();

    result.re = ZERO;
    result.im = ZERO;
    if(z2.re == ZERO && z2.im ==ZERO)
        {
        puts("ERROR: Divide by Zero");
        return(result);
        }
    temp = creal(cmult(z2,cconj(z2)));
    result = cmult(z1,cconj(z2));
    result.re = result.re / temp;
    result.im = result.im / temp;
    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cexp.c

        Calling Sequence:       cexp(z)

        Usage:  Performs the complex exponential function on
                a complex variable.

        Parameters:     z       COMPLEX
                        A COMPLEX variable.

        Return:         COMPLEX
                        The complex exponential of the argument.

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/
#include <math.h>
#include "amath.h"

COMPLEX cexp(z)
    COMPLEX z;
{
    COMPLEX result;
    result.re = exp(z.re) * cos(z.im);
    result.im = exp(z.re) * sin(z.im);
    return(result);
}
```

```
/*********************************************************

        Source File Name:       cexpon.c

        Calling Sequence:       cexpon(w);

        Usage:  This function evaluates e raised to the jw.

        Parameters:     w       double
                        The argument in rad.

                        cexpon()COMPLEX
                        The result.

        Author: Alan L. Ferguson

        Date:   10/16/87

*********************************************************/

#include <math.h>
#include "amath.h"

COMPLEX cexpon(w)
    double      w;
{
    COMPLEX     result;

    result.re = cos(w);
    result.im = sin(w);

    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cfft.c

        Calling Sequence:       cfft(data,n,flag)

        Usage:  This routine performs the FFT operation on
                the data pointed to by x.  It must be a
                power of two!!

        Parameters:     data[]  COMPLEX
                        A COMPLEX pointer addressing the
                        data array.

                        n       int
                        An integer with value equal to the
                        length of the data sequence.

                        flag    int
                        A flag denoting forward or inverse
                        transform.
                                1       Forward
                               -1       Inverse

        Return:         int
                        The number of iterations in process.

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/
#include "amath.h"              /*  Complex definitions       */
#include <math.h>
#include <stdio.h>

int cfft(data,n,flag)
    COMPLEX     data[];
    int         flag,n;
{
    COMPLEX     gtemp,htemp,w,temporary;
    int         i,j,iteration,m,no_iterations,n_points,offset,
                number,result,temp,power,r_power,rev;
    double      multiplier, sign;

/*  Check for the number of iterations                        */
    number = 1;
    no_iterations = 0;
    while(number < n)
        {
        number = number * 2;
        no_iterations++;
        }
```

```
/*  Determine type of transform to perform                         */
    if(flag == FORWARD)
        {
        multiplier = (double) 1/number;
        sign = -1.0;
        }

    if(flag == INVERSE)
        {
        multiplier = (double) ONE;
        sign = 1.0;
        }

/*  Begin FFT process by method of frequency decomposition         */
    result = no_iterations;
    number = pow((double)TWO,(double)no_iterations);
    offset = number;
    n_points = number;

    for(iteration=1; iteration<= no_iterations; iteration++)
        {
        offset /= 2;

        for(j=0; j < number; j+=n_points)
            {
            w = cmplx(cos(2.0*PI/(double)n_points),
                sign*sin(2.0*PI/(double)n_points));
            for(m=0; m<offset; m++)
                {
                gtemp = cadd(data[m+j],data[m+j+offset]);
                htemp = cmult((csub(data[m+j],data[m+j+offset])),
                              (cpow(w,(double)m)));
                data[m+j] = gtemp;
                data[m+j+offset] = htemp;
                }
            }
        n_points /=2;
        }

/*  Swap the bit-reversed coeffecients                             */
    n_points = number/2;
    temp = 1;

    for (i=1; i<number; i++)
        {
        if(i < temp)
            {
            temporary = data[temp-1];
            data[temp-1] = data[i-1];
            data[i-1] = temporary;
            }
        for(j = n_points; j<temp; j/=2)
            temp -=j;
```

```
        temp +=j;
        }

/*  Scale by multiplier                                              */
    for(i=0; i<=number-1; i++)
        {
        data[i].re = data[i].re * multiplier;
        data[i].im = data[i].im * multiplier;
        }

    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cfilter.c

        Calling Sequence:       cfilter(f, filtkey, f3)

        Usage:  This routine scales a variable with the
                scaling being dependent where in a certain
                filter the frequency occurs.

        Parameters:     f       double
                        The frequency of interest.  The
                        magnitude.

                        filtkey int
                        A key refering to what type of
                        filter is to be used.

                        f3      double
                        The 3 dB frequency of the filter of
                        interest.

*****************************************************************

        The following filters are available:

        Key     Filter Type
        ___     ___________
        1        1-Pole Butterworth
        2        2-Pole Butterworth
        3        3-Pole Butterworth
        4        4-Pole Butterworth
        5        5-Pole Butterworth
        6        6-Pole Butterworth
        7        7-Pole Butterworth
        8        8-Pole Butterworth

        9        SAW Filter

        10       T-second Integrator (pass T as F3)

        11       2 -Pole Chebyshev      3.00 dB Ripple
        12       2 -Pole Chebyshev      1.00 dB Ripple
        13       2 -Pole Chebyshev      0.10 dB Ripple
        14       2 -Pole Chebyshev      0.01 dB Ripple
        15       3 -Pole Chebyshev      3.00 dB Ripple
        16       3 -Pole Chebyshev      1.00 dB Ripple
        17       3 -Pole Chebyshev      0.10 dB Ripple
        18       3 -Pole Chebyshev      0.01 dB Ripple
        19       4 -Pole Chebyshev      3.00 dB Ripple
        20       4 -Pole Chebyshev      1.00 dB Ripple
        21       4 -Pole Chebyshev      0.10 dB Ripple
        22       4 -Pole Chebyshev      0.01 dB Ripple
```

```
        23       5 -Pole Chebyshev      3.00 dB Ripple
        24       5 -Pole Chebyshev      1.00 dB Ripple
        25       5 -Pole Chebyshev      0.10 dB Ripple
        26       5 -Pole Chebyshev      0.01 dB Ripple
        27       6 -Pole Chebyshev      3.00 dB Ripple
        28       6 -Pole Chebyshev      1.00 dB Ripple
        29       6 -Pole Chebyshev      0.10 dB Ripple
        30       6 -Pole Chebyshev      0.01 dB Ripple
        31       7 -Pole Chebyshev      3.00 dB Ripple
        32       7 -Pole Chebyshev      1.00 dB Ripple
        33       7 -Pole Chebyshev      0.10 dB Ripple
        34       7 -Pole Chebyshev      0.01 dB Ripple
        35       8 -Pole Chebyshev      3.00 dB Ripple
        36       8 -Pole Chebyshev      1.00 dB Ripple
        37       8 -Pole Chebyshev      0.10 dB Ripple
        38       8 -Pole Chebyshev      0.01 dB Ripple

        39       2 -Pole Linear Phase(1 sec delay at w=0)
        40       3 -Pole Linear Phase(1 sec delay at w=0)
        41       4 -Pole Linear Phase(1 sec delay at w=0)
        42       5 -Pole Linear Phase(1 sec delay at w=0)
        43       6 -Pole Linear Phase(1 sec delay at w=0)
        44       7 -Pole Linear Phase(1 sec delay at w=0)
        45       8 -Pole Linear Phase(1 sec delay at w=0)

*****************************************************************

        Author: Alan L. Ferguson

        Date:   8/25/87

*****************************************************************/
#include <math.h>
#include <stdio.h>
#include "amath.h"


COMPLEX cfilter(f,filtkey,f3)
    double      f, f3;
    int         filtkey;
{
    int i;
    COMPLEX     jw, a[12], result, temp1, temp2;

/*  Set jw = s                                                   */
    jw = cmplx(ZERO,(double)f/f3);

/*  Set the default value to Zero                                */
    result.re = 0.0;
    result.im = 0.0;
```

```
/*  Zero the imaginary part of the coeffecients                    */
    for(i=0; i<12; i++)
        a[i].im = 0.0;

/*  Choose the appropriate filter                                  */
    switch (filtkey)
    {
        case 1:
                a[0].re =1.;
                a[1].re =0.;
                a[2].re =0.;
                a[3].re =1.;
                a[4].re =1.;
                a[5].re =0.;
                a[6].re =0.;
                a[7].re =0.;
                a[8].re =0.;
                a[9].re =0.;
                a[10].re =0.;
                a[11].re =0.;
                break;

        case 2:
                a[0].re =1.;
                a[1].re =0.;
                a[2].re =0.;
                a[3].re =1.;
                a[4].re =1.41421;
                a[5].re =1.;
                a[6].re =0.;
                a[7].re =0.;
                a[8].re =0.;
                a[9].re =0.;
                a[10].re =0.;
                a[11].re =0.;
                break;

        case 3:
                a[0].re =1.;
                a[1].re =0.;
                a[2].re =0.;
                a[3].re =1.;
                a[4].re =2.;
                a[5].re =2.;
                a[6].re =1.;
                a[7].re =0.;
                a[8].re =0.;
                a[9].re =0.;
                a[10].re =0.;
                a[11].re =0.;
                break;
```

```
case 4:
        a[0].re =1.;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.;
        a[4].re =2.6131;
        a[5].re =3.4142;
        a[6].re =2.6131;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 5:
        a[0].re =1.;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.;
        a[4].re =3.2361;
        a[5].re =5.2361;
        a[6].re =5.2361;
        a[7].re =3.2361;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 6:
        a[0].re =1.;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.;
        a[4].re =3.8637;
        a[5].re =7.4641;
        a[6].re =9.1416;
        a[7].re =7.4641;
        a[8].re =3.8637;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 7:
        a[0].re =1.;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.;
        a[4].re =4.4940;
        a[5].re =10.0978;
```

```
        a[6].re =14.5918;
        a[7].re =14.5918;
        a[8].re =10.0978;
        a[9].re =4.4940;
        a[10].re =1.;
        a[11].re =0.;
        break;

case 8:
        a[0].re =1.;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.;
        a[4].re =5.1258;
        a[5].re =13.1371;
        a[6].re =21.8462;
        a[7].re =25.6884;
        a[8].re =21.8462;
        a[9].re =13.1371;
        a[10].re = 5.1258;
        a[11].re =1.;
        break;

case 11:
        a[0].re =.50062;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.70715;
        a[4].re =.64452;
        a[5].re =1.;
        a[6].re =0.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 12:
        a[0].re =.66276;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.74363;
        a[4].re =.90151;
        a[5].re =1.;
        a[6].re =0.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;
```

```
case 13:
        a[0].re =.86760;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.87765;
        a[4].re =1.22081;
        a[5].re =1.;
        a[6].re =0.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 14:
        a[0].re =.95420;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.95530;
        a[4].re =1.34868;
        a[5].re =1.;
        a[6].re =0.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 15:
        a[0].re =.25035;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.25035;
        a[4].re =.92774;
        a[5].re =.59706;
        a[6].re =1.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 16:
        a[0].re =.37429;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.37429;
        a[4].re =1.03303;
        a[5].re =.90268;
```

```
        a[6].re =1.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 17:
        a[0].re =.61123;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.61123;
        a[4].re =1.36286;
        a[5].re =1.39582;
        a[6].re =1.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 18:
        a[0].re =.78718;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.78718;
        a[4].re =1.64659;
        a[5].re =1.69337;
        a[6].re =1.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 19:
        a[0].re =.1252;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.1769;
        a[4].re =.4046;
        a[5].re =1.1689;
        a[6].re =.5812;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;
```

```
case 20:
        a[0].re =.1998;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.2242;
        a[4].re =.6360;
        a[5].re =1.3112;
        a[6].re =.9049;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 21:
        a[0].re =.3782;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.3826;
        a[4].re =1.1346;
        a[5].re =1.7850;
        a[6].re =1.4869;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 22:
        a[0].re =.5622;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.5629;
        a[4].re =1.5990;
        a[5].re =2.2936;
        a[6].re =1.9125;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 23:
        a[0].re =.06261;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.06261;
        a[4].re =.4078;
        a[5].re =.5488;
```

```
        a[6].re =1.4147;
        a[7].re =.5745;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 24:
        a[0].re =.1040;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.1040;
        a[4].re =.5083;
        a[5].re =.8820;
        a[6].re =1.5803;
        a[7].re =.9062;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 25:
        a[0].re =.2177;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.2177;
        a[4].re =.8660;
        a[5].re =1.6407;
        a[6].re =2.1520;
        a[7].re =1.5369;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 26:
        a[0].re =.3627;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.3627;
        a[4].re =1.3347;
        a[5].re =2.4383;
        a[6].re =2.8469;
        a[7].re =2.0480;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;
```

```
case 27:
        a[0].re =.031305;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.044219;
        a[4].re =.16335;
        a[5].re =.698804;
        a[6].re =.69044;
        a[7].re =1.66249;
        a[8].re =.57068;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 28:
        a[0].re =.053455;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.059978;
        a[4].re =.27353;
        a[5].re =.85633;
        a[6].re =1.12151;
        a[7].re =1.84353;
        a[8].re =.90698;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 29:
        a[0].re =.12015;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.12154;
        a[4].re =.57829;
        a[5].re =1.43530;
        a[6].re =2.12876;
        a[7].re =2.48288;
        a[8].re =1.56658;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 30:
        a[0].re =.21859;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.21884;
        a[4].re =.99163;
        a[5].re =2.25965;
```

```
        a[6].re =3.25896;
        a[7].re =3.31983;
        a[8].re =2.13412;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 31:
        a[0].re =.015660;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.015660;
        a[4].re =.14614;
        a[5].re =.29999;
        a[6].re =1.05175;
        a[7].re =.83139;
        a[8].re =1.91147;
        a[9].re =.5684;
        a[10].re =1.;
        a[11].re =0.;
        break;

case 32:
        a[0].re =.027253;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.027253;
        a[4].re =.19291;
        a[5].re =.50381;
        a[6].re =1.26811;
        a[7].re =1.35757;
        a[8].re =2.10317;
        a[9].re =.90750;
        a[10].re =1.;
        a[11].re =0.;
        break;

case 33:
        a[0].re =.064585;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.064585;
        a[4].re =.37852;
        a[5].re =1.06716;
        a[6].re =2.07911;
        a[7].re =2.60152;
        a[8].re =2.79095;
        a[9].re =1.58543;
        a[10].re =1.;
        a[11].re =0.;
        break;
```

```
case 34:
        a[0].re =.12595;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.12595;
        a[4].re =.68572;
        a[5].re =1.85737;
        a[6].re =3.29509;
        a[7].re =4.04874;
        a[8].re =3.73515;
        a[9].re =2.19127;
        a[10].re =1.;
        a[11].re =0.;
        break;

case 35:
        a[0].re =.0078288;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.011058;
        a[4].re =.056474;
        a[5].re =.32070;
        a[6].re =.47185;
        a[7].re =1.46650;
        a[8].re =.97189;
        a[9].re =2.16057;
        a[10].re =.56696;
        a[11].re =1.;
        break;

case 36:
        a[0].re =.013831;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.015519;
        a[4].re =.097971;
        a[5].re =.41410;
        a[6].re =.79334;
        a[7].re =1.74349;
        a[8].re =1.59161;
        a[9].re =2.36061;
        a[10].re =.90788;
        a[11].re =1.;
        break;

case 37:
        a[0].re =.034141;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.034536;
        a[4].re =.22902;
        a[5].re =.78720;
```

```
        a[6].re =1.67634;
        a[7].re =2.79177;
        a[8].re =3.06245;
        a[9].re =3.08426;
        a[10].re =1.59803;
        a[11].re =1.;
        break;

case 38:
        a[0].re =.070308;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =.070308;
        a[4].re =.44639;
        a[5].re =1.42188;
        a[6].re =2.93459;
        a[7].re =4.41480;
        a[8].re =4.80689;
        a[9].re =4.10960;
        a[10].re =2.23073;
        a[11].re =1.;
        break;

case 39:
        a[0].re =1.61804;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =1.61804;
        a[4].re =2.20321;
        a[5].re =1.;
        a[6].re =0.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 40:
        a[0].re =2.7718;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =2.7718;
        a[4].re =4.86637;
        a[5].re =3.41750;
        a[6].re =1.;
        a[7].re =0.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;
```

```
case 41:
        a[0].re =5.25828;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =5.25828;
        a[4].re =11.11552;
        a[5].re =10.07023;
        a[6].re =4.73057;
        a[7].re =1.;
        a[8].re =0.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 42:
        a[0].re =11.21331;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =11.21331;
        a[4].re =27.21909;
        a[5].re =29.36504;
        a[6].re =17.82010;
        a[7].re =6.17948;
        a[8].re =1.;
        a[9].re =0.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 43:
        a[0].re =26.6313;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =26.6313;
        a[4].re =71.9941;
        a[5].re =88.4667;
        a[6].re =63.7755;
        a[7].re =28.7348;
        a[8].re =7.7681;
        a[9].re =1.;
        a[10].re =0.;
        a[11].re =0.;
        break;

case 44:
        a[0].re =69.2265;
        a[1].re =0.;
        a[2].re =0.;
        a[3].re =69.2265;
        a[4].re =204.3353;
        a[5].re =278.3697;
```

```
                a[6].re =228.2392;
                a[7].re =122.4894;
                a[8].re =43.3861;
                a[9].re =9.48609;
                a[10].re =1.;
                a[11].re =0.;
                break;

        case 45:
                a[0].re =194.054;
                a[1].re =0.;
                a[2].re =0.;
                a[3].re =194.054;
                a[4].re =617.007;
                a[5].re =915.511;
                a[6].re =831.692;
                a[7].re =508.541;
                a[8].re =215.592;
                a[9].re =62.3170;
                a[10].re =11.3223;
                a[11].re =1.;
                break;

        case 9:
                result.re = (1./.54)*sin(.2264*PI*f/f3)/
                            (.2264*PI*f/f3)*
                            (.54*sin(.566*PI*f/f3)/
                            (.566*PI*f/f3)+
                            .23*sin(PI+.566*PI*f/f3)/
                            (PI+.566*PI*f/f3)+
                            .23*sin(.566*PI*f/f3-PI)/
                            (.566*PI*f/f3-PI));
                return(result);

        case 10:
                result = cmult(cmplx(sin(PI*f*f3)/
                            (PI*f*f3),0.),
                            cmplx(cos(PI*f*f3),
                            -sin(PI*f*f3)));
                return(result);

        default:
                puts("Selected filter not available");
                return(result);

    }

/*****************************************************************/
/*  Polynomial Transfer Function                                 */
/*****************************************************************/
```

```
/*  Evaluate numerator                                          */
    temp1 = a[0];
    for(i=1; i<3; i++)
        temp1 = cadd(temp1,cmult(a[i],cpow(jw,(double)i)));

/*  Evaluate denominator                                        */
    temp2 = a[3];
    for (i=4; i<12; i++)
        temp2 = cadd(temp2,cmult(a[i],cpow(jw,
                    (double)(i-3))));

/*  Calculate result                                            */
    result = cdiv(temp1,temp2);

    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cmag.c

        Calling Sequence:       cmag(z)

        Usage:  This routine calculates the absolute value
                of a COMPLEX number.

        Parameters:     z       COMPLEX
                        The number to find the absolute
                        value of.

        Return:         double
                        The absolute value of the argument.

        Author: Alan L. Ferguson

        Date:   12/06/87

*****************************************************************/

#include "amath.h"
#include <math.h>

double cmag(z)
    COMPLEX z;
{
    double result;

    result = sqrt(z.re*z.re + z.im*z.im);
    return(result);
}
```

```
/*****************************************************************

        Source File Name:       cmplx.c

        Calling Sequence:       cmplx(x,y);

        Usage:  This routine creates a complex variable from
                two double variables.

        Parameters:     x,y     double
                        The real and imaginary parts.

        Return:         COMPLEX
                        A complex variable of the two doubles.

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/
#include "amath.h"

COMPLEX cmplx(x,y)
    double x,y;
{
    COMPLEX result;

    result.re = x;
    result.im = y;
    return(result);
}
```

```
/****************************************************************

        Source File Name:       cmult.c

        Calling Sequence:       cmult(z1,z2)

        Usage:  This routine performs a COMPLEX multiplica-
                tion on z1 and z2.

        Parameters:     z1,z2   COMPLEX
                        The COMPLEX numbers to be multiplied

        Return:         COMPLEX
                        The resulting complex multiplication.
        Author: Alan L. Ferguson

        Date:   11/16/87

****************************************************************/

#include "amath.h"

COMPLEX cmult(z1,z2)
    COMPLEX z1,z2;
{
    COMPLEX result;

    result.re = (z1.re * z2.re) - (z1.im * z2.im);
    result.im = (z1.im * z2.re) + (z1.re * z2.im);

    return(result);
}
```

```
/****************************************************************

        Source File Name:       cpow.c

        Calling Sequence:       cpow(z,n)

        Usage:  This routine evaluates the power of a
                COMPLEX number.

        Parameters:     z       COMPLEX
                        The argument to be taken to a power.

                        n       int
                        An integer indicating the power.

        Return:         COMPLEX
                        The resulting complex value.

        Author: Alan L. Ferguson

        Date:   11/16/87

****************************************************************/
#include "amath.h"              /*  Complex definitions        */
#include <math.h>

COMPLEX cpow(z,n)
    COMPLEX z;
    double n;
{
    COMPLEX result;
    double scale, theta;

/*  Check for exponent of ZERO                                  */
    if(n == ZERO)
        {
        result.re = 1.0;
        result.im = 0.0;
        return(result);
        }

    scale = pow(ccabs(z), n);
    if(z.re == ZERO)
        if(z.im < ZERO)
            theta = - PI / TWO;
        else
            theta = PI / TWO;
    else
        theta = atan2( z.im, z.re);

    result.re = scale * cos(n * theta);
    result.im = scale * sin(n * theta);
    return(result);
}
```

```
/*****************************************************************

        Source File Name:       csub.c

        Calling Sequence:       csub(z1,z2)

        Usage:  This routine subtracts z2 from z1.

        Parameters:     z1,z2   COMPLEX
                        The numbers to be subtracted.

        Return:         COMPLEX
                        The resulting complex subtraction.

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/

#include "amath.h"

COMPLEX csub(z1,z2)
    COMPLEX z1,z2;
{
    COMPLEX result;

    result.re = z1.re - z2.re;
    result.im = z1.im - z2.im;
    return(result);
}
```

```
/****************************************************************

        Source File Name:       creal.c

        Calling Sequence:       creal(z)

        Usage:  This routine takes the real value of a
                COMPLEX number.

        Parameters:     z       COMPLEX
                        The complex number of interest.

        Return:         double
                        The real part of the argument.

        Author: Alan L. Ferguson

        Date:   11/16/87

****************************************************************/
#include "amath.h"

double creal(z)
    COMPLEX z;
{
    double result;

    result = z.re;
    return(result);
}
```

```
/*****************************************************************

        Source File Name:       sinc.c

        Calling Sequence:       sinc(x)

        Usage:  This routine calculates the sin x / x
                function.

        Parameters:     x       double
                        The argument of the function.

        Return:         double
                        The resulting sin x / x

        Author: Alan L. Ferguson

        Date:   11/16/87

*****************************************************************/
#include <math.h>
#include "amath.h"

double sinc(x)
    double x;
{
    double result;

    if (x == ZERO)
        result = ONE;
    else
        result = sin(x) / x;

    return(result);
}
```

```
/***************************************************************
        Source File Name:       window.c

        Calling Sequence:       window(type, f)

        Usage:  This function returns the window function evalu-
                ated at the frequency of interest (f) for either
                a Gaussian aperature or a rectangular aperature.

        Parameters:             f       (input) double
                                The frequency of interest.

                                type    (input) int
                                Type of aperature for Bragg cell

                                        0 - Gaussian
                                        1 - rectangular

        Return:                 COMPLEX
                                The window function at f.

        Author: Alan L. Ferguson

        Date:   5/27/1988

***************************************************************/
#include <math.h>
#include "amath.h"              /* Complex definitions          */

COMPLEX window(type, f)
    int         type;
    double      f;
{
    double      alpha2 = 36.966e12;
    COMPLEX     result;

/*  Gaussian window in time  250 ns                             */
    if(type == 0)
        {
        result.re = exp( - (PI*PI * f*f) / alpha2);
        result.im = ZERO;
        }
    else

/*  Rectangular window in time  250 ns                          */
        {
        result.re = sinc(2.0*PI * f * 250.0e-9);
        result.im = ZERO;
        }

/*  Return window value at given frequency                      */
    return(result);

}
```

AN ANALYSIS OF THE INTERFEROMETRIC ACOUSTO–OPTIC SPECTRUM ANALYZER

by

ALAN LEWIS FERGUSON

B.S., Kansas State University, 1986

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Abstract

A mathematical model and a computer algorithm for predicting the performance of the interferometric acousto–optic (IFAO) spectrum analyzer are provided. The work of Vander Lugt is reviewed and made more general in nature, allowing for more generalized input and reference signal waveforms. After a generalized representation for the analyzer is derived, a computer algorithm is developed. The algorithm is the basis for a computer program written to emulate the analyzer using coherent detection as the means for detecting the signal. The results of the program for several test cases are analyzed and compared to the theoretical results. Several conclusions are drawn from the computer results that should be taken into consideration when implementing the IFAO spectrum analyzer in hardware.