SIMULATION OF NON–IDEALITIES IN FREQUENCY COMPRESSIVE RECEIVER COMPONENTS AND INPUTS

by

PHILLIP EDWARD FRY

B.S., Kansas State University, 1987

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

ELECTRICAL ENGINEERING

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:
Brian Harms
Major Professor

Table of Contents

List of Figures

Acknowledgements

I would like to thank the following people and institutions for their support:

Dr. Brian Harms

Peggy Miles

Ed and Lila Fry

Cam Ingelin

Dept. of Electrical and Computer Engineering, Kansas State University

Motorola Government Electronics Group

First National Bank, Hamilton, Kansas

I. Introduction

The purpose of this report is to present the results of simulations performed on a frequency compressive receiver. The simulations deal with how unmatched Sweeping Local Oscillators (S.L.O.) and Dispersive Filters affect performance. Also examined is how three types of non–ideal input affect performance. Performance criteria examined were probability of detection, spectral resolution and frequency determination.

A frequency compressive receiver detects signals by rapidly scanning, or tuning, the frequency band of interest. It is similar to a spectrum analyzer in that both are Fourier Transform receivers. However, the frequency compressive receiver scans faster over larger bands than does the spectrum analyzer. It is this characteristic which gives the frequency compressive receiver a higher probability of intercept (P.O.I.) than the spectrum analyzer.

In order to understand the frequency compressive receiver, it is instructive to examine first how a spectrum analyzer, or scanning receiver, works. Figure 1 shows a block diagram of a simple spectrum analyzer. The bandpass filter admits only signals in the band of interest into the analyzer. The S.L.O. generates a sinusoid whose frequency is a linearly increasing function of time. The mixer output is a sum and difference term from its two inputs. The Intermediate Frequency (I.F.) filter is a narrow bandpass filter which only passes the difference term from the mixer. This filter determines the spectral resolution of the analyzer.

With the input signal into the mixer being continuous wave (C.W.), the mixer difference term is a sinusoid whose frequency is linearly decreasing with time. As the sinusoid's difference frequency sweeps through the passband of the I.F. filter, the filter's output is envelope detected (magnitude squared).

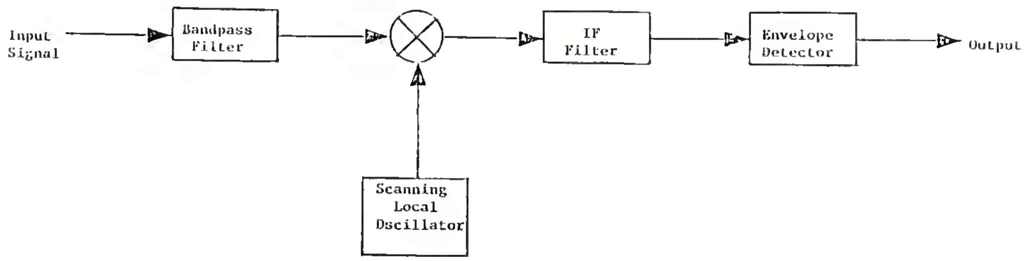


Figure I. Spectrum Analyzer Block Diagram.

This magnitude is the magnitude of the spectral component at that input frequency. The S.L.O. can be calibrated in frequency to give the frequency of the spectral component by the position in the scan time. Thus, the spectral components are sorted in time, and an approximation of the magnitude Fourier transform of the frequency band is formed.

The rate at which the S.L.O. scans is limited by the bandwidth, or resolution, of the I.F. filter. If the S.L.O. scans the bandwidth of the I.F. filter faster than its impulse response, the filter output is impulsive. However, reducing the scan rate of the S.L.O. reduces the P.O.I. for signals of duration less than the scan rate.

Unfortunately, the better the spectral resolution, the slower the S.L.O. must sweep. An approximate relationship is given by the equation [1],

$$S \approx B^2 \tag{1}$$

where S is the scan rate in Hz/seconds, and B is the I.F. filter bandwidth in Hz. For narrow band signals, high spectral resolution is desirable. Thus, the problem is how to scan wide bandwidths with high spectral resolution rapidly.

The frequency compressive receiver overcomes these limitations of the spectrum analyzer, at least theoretically. The receiver's theoretical basis comes from the Chirp transform's multiply–convolve–multiply implementation. It can also be thought of as a matched filter receiver where the filter is matched to a long C.W. signal.

The key to the frequency compressive receiver is the dispersive filter which immediately follows the mixer. (See Fig. 2.) It is characterized by a linear time delay for spectral componeuts as a function of frequency. The

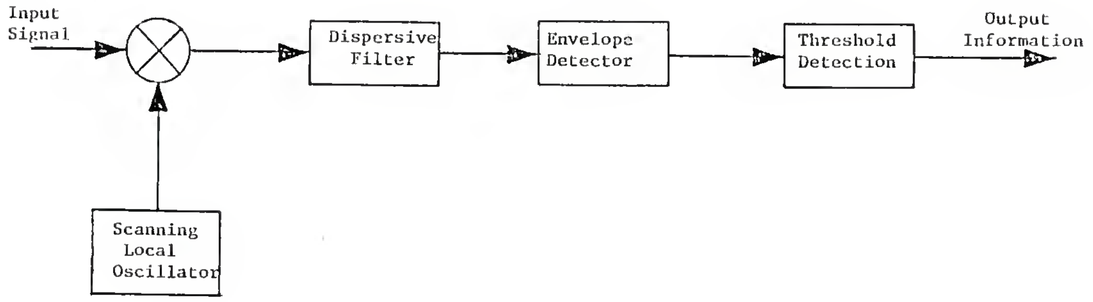


Figure 2. Frequency Compressive Receiver Block Diagram

range of frequencies for which this relationship holds is called the compressive bandwidth. The delay difference between the highest and lowest frequencies is called the dispersion time.

The S.L.O. is matched by making its phase the complex conjugate of the dispersive filter's phase. Representing the S.L.O. output as

$$q(t) = e^{-j\mu t^2} \tag{2}$$

the output from the dispersive filter for an input signal s(t) is

$$\begin{aligned} s(t)\, e^{-j\mu t^2} * e^{j\mu t^2} &= \int s(\tau)\, e^{-j\mu\tau^2}\, e^{j\mu(t-\tau)^2}\, d\tau \\ &= e^{j\mu t^2} \int s(\tau)\, e^{-j2\mu t\tau}\, d\tau \end{aligned} \tag{3}$$

The final expression is the Fourier transform of s(t) with a phase factor. If only the magnitude transform is of interest, the final multiplication can be omitted. Thus, the Fourier transform of the input is generated.

Figure 3 details the receiver operation more explicitly. The C.W. signal s(t) is mixed with the S.L.O. Because the signal has a finite bandwidth, the effect of the S.L.O. is to "chirp" the spectral components of s(t). The mixer produces a sum and difference term from its inputs. The slope of the difference term matches that of the dispersive filter. Therefore, all the energy from s(t) spectral components arrive at the output of the dispersive filter simultaneously. It compresses the signal. The sum term from the mixer does not have the correct slope, so its spectral terms are dispersed in time as they traverse the filter. Thus, their contribution to the output can be ignored.

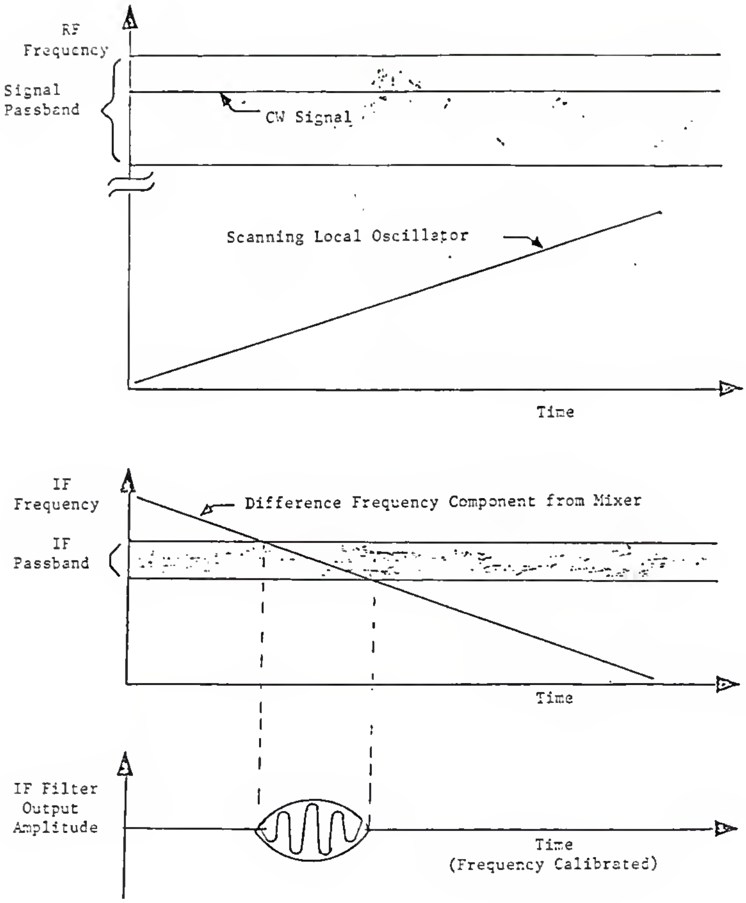

RF Frequency
Signal Passband
CW Signal
Scanning Local Oscillator
Time
IF Frequency
Difference Frequency Component from Mixer
IF Passband
Time
IF Filter Output Amplitude
Time (Frequency Calibrated)

Figure 3. Frequency-Time Diagram for Frequency Compressive Receiver

The impulsive output forms the Fourier transform of the dispersive filter's magnitude response. This magnitude response determines the spectral resolution of the frequency compressive receiver. Narrowing this response improves the receiver's spectral resolution. This can be done by widening the bandwidth of the dispersive filter. Widening the bandwidth of the filter allows the S.L.O. to sweep much faster, because the impulse response is shorter for a wideband filter. It also allows the receiver to cover a larger slice of the spectrum. To summarize, the greater the spectral resolution, the faster the scan is allowed to sweep. This is exactly opposite to the relation for the spectrum analyzer.

To fully compress signals which lie near the edge of the compressive bandwidth, the S.L.O. must sweep over twice the range to insure that the signal will be fully compressed by the dispersive filter. This also means that the time required to make one scan of the band is twice the dispersion time.

Input signals which last longer than twice the dispersion time are "gated" by the S.L.O. to appear to last only one sweep time. Pulses of shorter duration than one sweep time suffer from reduced compression.

II. Development

This section briefly develops the frequency compressive receiver's mathematical model and implementation. A more thorough development can be found in [1], [2] and [3]. The model was implemented on a Digital Electronic Corp. VAX 11/750 using VAX C.

The equivalent low-pass model was used, so only the passband of the receiver need be represented instead of the signal carrier. This greatly reduces the number of data points required to represent the signal completely. The problem of signal aliasing is avoided if the restriction is observed [2],

$$\tau \leq \frac{N}{4\ B_w} \tag{1}$$

where N is the number of data points, B_w is the compressive bandwidth, and τ is the dispersion time.

The key to the frequency compressive receiver is the matching of S.L.O. phase response to the complex conjugate of the dispersive filter's phase response. For the S.L.O., frequency is a linear function of time. Therefore, f(t) must satisfy the following equations.

$$f(t)\Big|_{t\,=\,0} = a\,t + b = -B_w \tag{2}$$

$$b = -B_w$$

$$f(t)\Big|_{t\,=\,\tau} = a\,t + b = 0 \tag{3}$$

$$a\,\tau + b = 0$$

$$f(t)\Big|_{t\,=\,2\tau} = a\,t + b = B_w \tag{4}$$

$$a\,2\tau + b = B_w$$

Simultaneously solving these equations gives the coefficients needed. To obtain the S.L.O.'s phase response, f(t) is integrated with respect to time to yield $\phi(t)$.

For the filter, time delay is a linear function of frequency. Therefore, $t_d(\omega)$ must satisfy the conditions

$$t_d(\omega)\Big|_{\omega\,=\,-\frac{B_w}{2}} = a\,\omega + b = -\frac{\tau}{2} \tag{5}$$

$$= -a\frac{B_w}{2} + b = -\frac{\tau}{2}$$

$$t_d(\omega)\Big|_{\omega\,=\,0} = a\,\omega + b = 0 \tag{6}$$

$$b = 0$$

$$t_d(\omega)\Big|_{\omega\,=\,\frac{B_w}{2}} = a\,\omega + b = \frac{\tau}{2} \tag{7}$$

$$= a\frac{B_w}{2} + b = \frac{\tau}{2}$$

$t_d(\omega)$ is also integrated to give the phase response of the dispersive filter. Then a negative sign must be multiplied through to form the complex conjugate. Note that the negative frequencies and time delays are a consequence of using the low-pass model.

In the low-pass model, the input signal is represented as

$$s(t) \;=\; \mathrm{Re}\{\, \tilde{s}(t)\, e^{j\omega_c t} \,\}\,, \tag{8}$$

where ω_c is the carrier frequency of the envelope of s(t). The S.L.O. is represented as

$$q(t) \;=\; e^{-j\mu t^2}\,, \tag{9}$$

and the dispersive filter as

$$H(f) \;=\; \frac{1}{2}\,\tilde{H}(\, f - f_c \,) \;+\; \frac{1}{2}\,\tilde{H}(\, -f - f_c \,) \tag{10}$$

where f_c is the center frequency of the compressive filter.

Calculations are first performed for the signal with no noise present. The input signal, s(t), is multiplied by the S.L.O. in the time domain. The result is then Fourier transformed to the frequency domain. The spectral terms are multiplied point by point times the dispersive filter transfer function evaluated at the particular frequency. This result is inverse Fourier transformed back to the time domain, where it is envelope detected. Detection occurs when the signal exceeds a certain threshold voltage V_t, set by the signal-to-noise ratio and the false alarm rate. Frequency determination comes from where the signal peak occurred with respect to the scan cycle.

The noise input, n(t), is assumed to be bandpass, stationary and Gaussian. The low-pass equivalent of n(t) is Gaussian, stationary and complex. All the receiver operations up to the envelope detector are linear, so the mean of n(t) is scaled. The new variance can be found from

$$\sigma_w^2 = B_n N_o \tag{11}$$

$$B_n = \int_{-\infty}^{+\infty} | \tilde{H}(f) |^2 \, df = f_3(\pi/\ln 2)^{1/2} \tag{12}$$

$$| H(f) | = e^{-0.34667(f/f_3)^2} \tag{13}$$

B_n is the noise bandwidth, N_o the noise density, and f_3 is the 3 dB frequency of the dispersive filter. H(f) is the Gaussian magnitude response.

The probability of detection is given by

$$P_{det} = 1 - P(|\omega(t)|^2 < V_t) \tag{14}$$

where V_t is the voltage threshold for detection, and $\omega(t)$ is the complex output. In the case of no signal, the complex terms are identically distributed Gaussian noise terms with the variance of equation (11). Squaring the noise terms transforms the distribution into a Gamma distribution. Adding the terms results in an exponential distribution of the form

$$F_w(\omega) = 1 - e^{\omega/2\sigma_w^2} \tag{15}$$

To set the detection threshold voltage V_t, the false alarm rate (FAR) must be set. This is the number of times per second the voltage threshold will be exceeded when only noise is present. Once known, the probability of false alarm is given by

$$P_{F.A.} = \frac{F.A.R.}{2\ B_{lp}} \tag{16}$$

where B_{lp} is the low-pass equivalent bandwidth of the receiver.

The P_{far} is related to V_t by equation (17). $F_v(V)$ had previously been found in equation (17),

$$P_{F.A.} = \int_{-V_t}^{+\infty} P(v)\ dv = 1 - \int_{-\infty}^{V_t} P(v)\ dv = 1 - F_v(V_t) \tag{17}$$

so substituting and solving for V_t yields

$$V_t = -2\ \sigma_w^2\ \ln(P_{FA}) \tag{18}$$

This is threshold voltage for detection for a given false alarm rate.

Evaluation of the probability of detection equation (14) is carried out via an algorithm developed by Brennan. (See reference [4].) There is no known closed form for a circular, non-zero mean, bivariate distribution.

III. Results

Preliminaries

This section describes the results from the computer simulations. It is divided into the following parts:

1. Linear Errors
   A. Sweeping Local Oscillator
   B. Dispersive Filter

2. Non–linear Errors
   A. Sweeping Local Oscillator
   B. Dispersive Filter

3. Short Pulse Input
   A. Ideal Receiver
   B. Linear S.L.O. Error

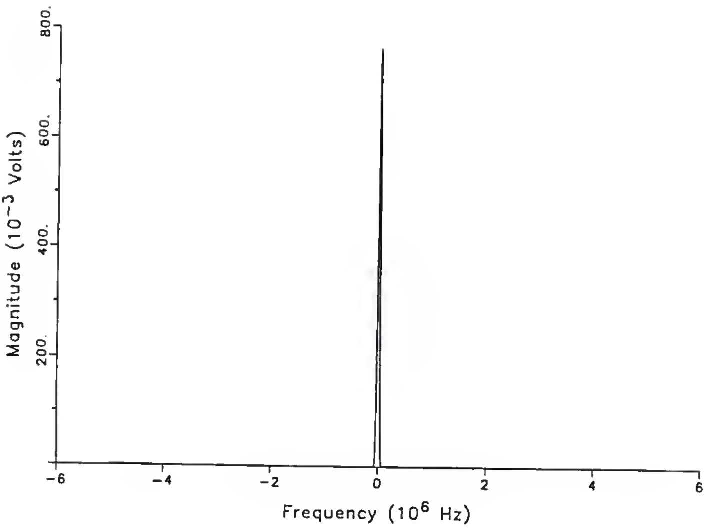
4. Long FSK Input

5. Long PSK Input

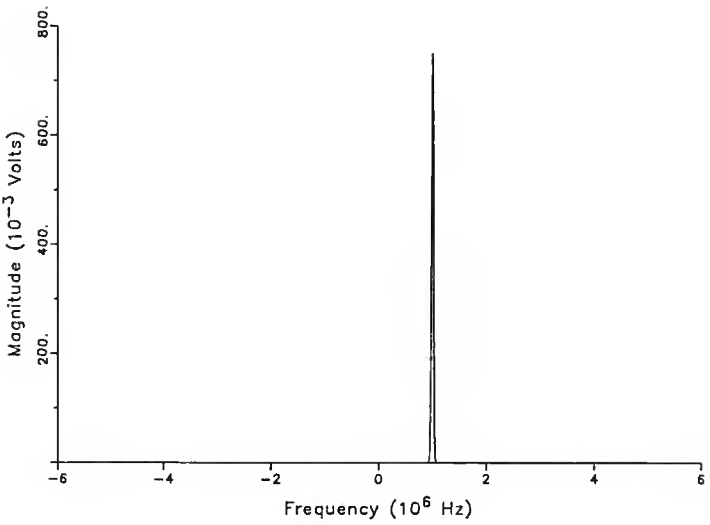
The receiver parameters used to generate the plots, unless otherwise stated, were: compressive bandwidth of 5 MHz, dispersion time of 40 microseconds, a sweeptime of 80 microseconds, and a 3 dB frequency of 1.8 MHz. The false alarm rate was 1 per second, and the signal to noise–density ratio was 58.3 dB. The input signal, for parts I and II was an 80 microsecond rectangular pulse, which is the ideal input for this receiver. The peak signal output values were scaled to the ideal or maximum value of the peak output. Only one part of the receiver was non–ideal at a time. The remaining parts were ideal.

The three performance criteria, probability of detection, frequency determination and spectral resolution degradation (SRD) are quantified in the following manner. The probability of detection is the likelihood that the peak of the output pulse will exceed threshold and be detected, given a certain signal to noise–density ratio and false–alarm rate. Frequency determination error is defined to be the difference between the receiver's estimate of the input frequency and its true value divided by the compressive bandwidth. This is also reported as a percentage. A negative sign indicates that the output signal peak was shifted to a lower frequency. Spectral resolution degradation is defined as a ratio of the non–ideal output, full width at half maximum (FWHM), divided by ideal output FWHM. It is reported as 10 * Log of this ratio. A logarithm was used because of the range of the ratios. The FWHM was graphically determined, so the SRD is only meant to give an idea of how much the main lobe widened with respect to the ideal.

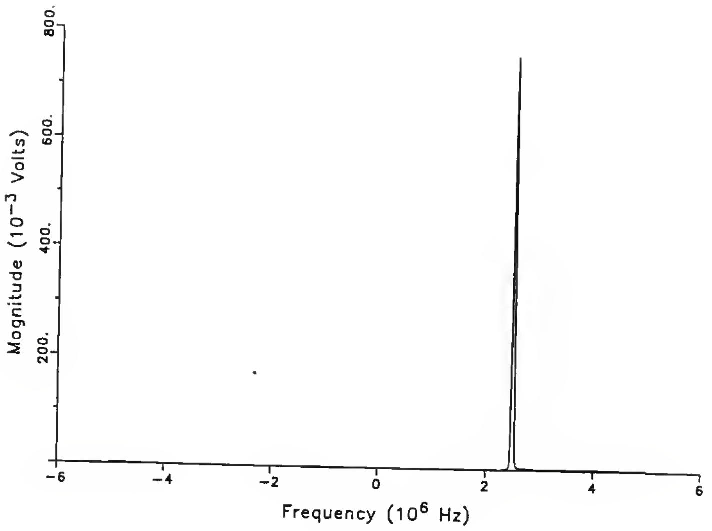
The first group of plots, 1–5, are for comparison with later plots. The S.L.O. and dispersive filter are ideal and the input pulse is long, hence it is an ideal input. What is varied in these plots is the frequency of the pulse in the passband. Values of 0, 1.0, 2.0, 2.5 and 3.0 MHz were used. This is to demonstrate how the output pulse changes as it is moved from the center of the passband to outside the passband. Note how the magnitude and shape stay the same until the pulse is moved outside the compression bandwidth (± 2.5 MHz). Then the pulse doesn't receive full compression, and the magnitude suffers accordingly.



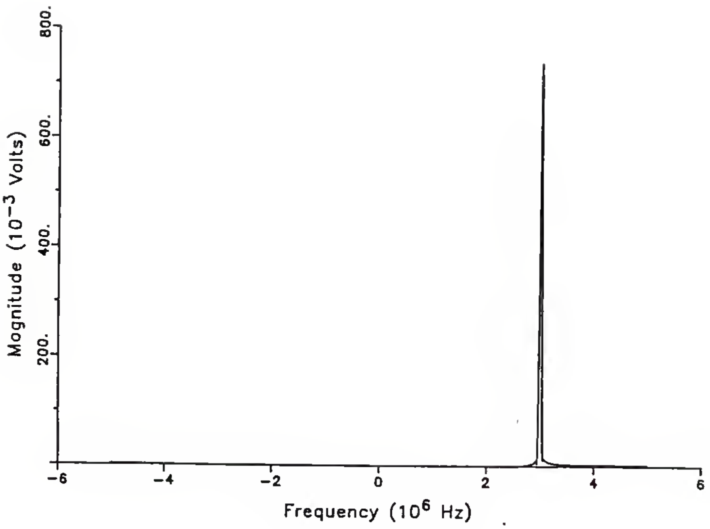
Plot 1 Ideal Frequency Compressive Receiver output with ideal input pulse at 0.0 MHz in the passband.



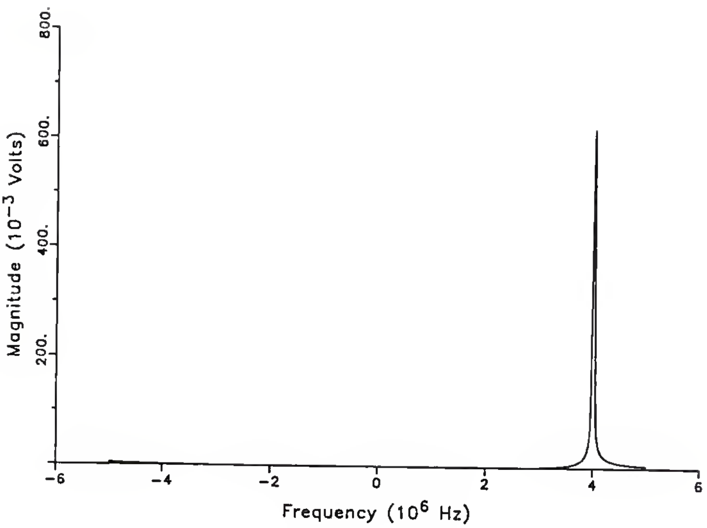
Plot 2 Ideal Frequency Compressive Receiver output with ideal input pulse at 1.0 MHz in the passband.



Plot 3 Ideal Frequency Compressive Receiver output with ideal input pulse at 2.5 MHz in the passband.



Plot 4 Ideal Frequency Compressive Receiver output with ideal input pulse at 3.0 MHz in the passband.



Plot 5 Ideal Frequency Compressive Receiver output with ideal input pulse at 4.0 MHz in the passband.

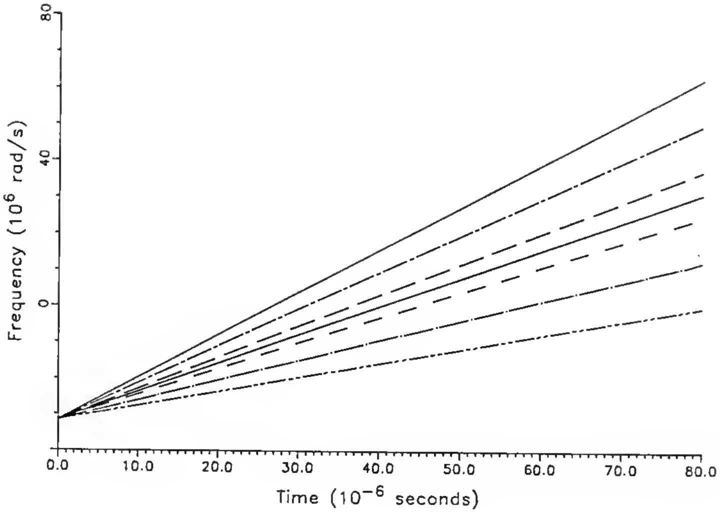
Section 1

In the theoretical development of the frequency compressive receiver, the frequency vs. time and time delay vs. frequency relationships for the SLO and dispersive filter, respectively, are assumed to be linear. In this section, linearity is maintained, but the slope of the function is deviated from the ideal by some factor. The non–ideal slope is reported as a factor times the ideal value.

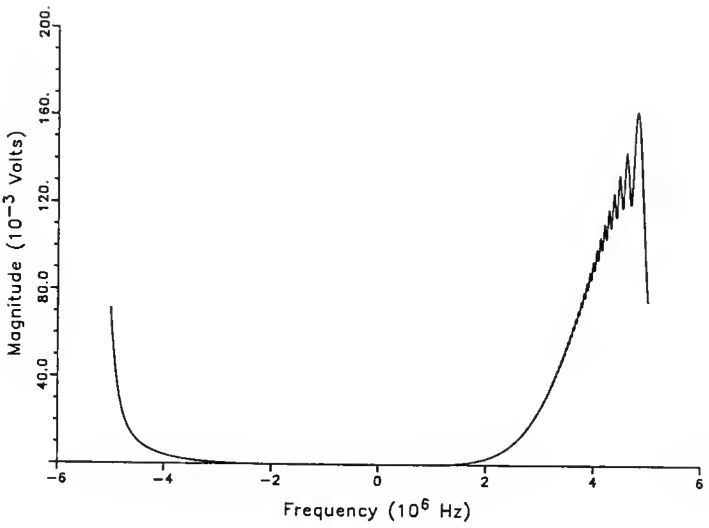
If the slope is multiplied by an error factor less than one, the S.L.O. scans too slowly. The output pulse is broadened and delayed, because the spectral components don't enter the dispersive filter quickly enough to all come out together. Hence, the output doesn't occur in an impulse but over a period of the scan. Because it is delayed, the peak is shifted to the right. Also, the S.L.O. doesn't scan the entire frequency range.

If the error multiplier is greater than one, the S.L.O. scans too rapidly. Now the spectral components enter the filter too soon with respect to the previous frequencies. Hence, the output occurs over a period of time much greater than an impulse. This has the same effect as scanning too slowly, except the output pulse is shifted to the left, or lower in frequency. Also, the S.L.O. now scans more than twice the compression bandwidth in the sweeptime.

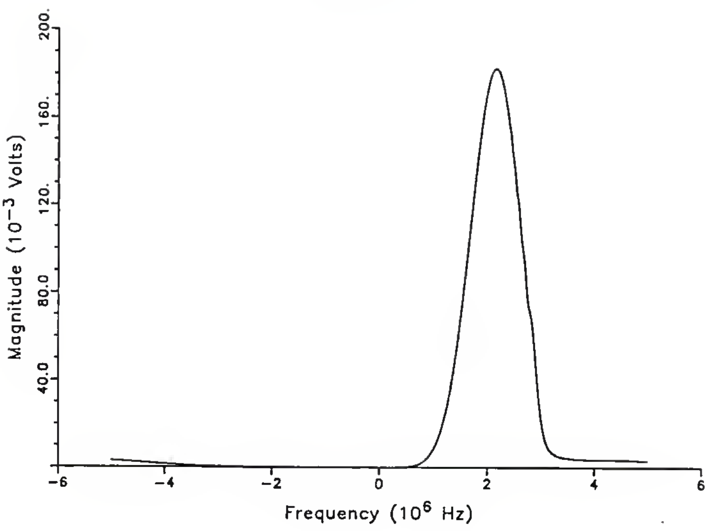
Plots 1A 2–7 show the output of the receiver with S.L.O. error multipliers of 0.50, 0.70, 0.90, 1.10, 1.30, and 1.50 respectively. Plot 1A 1 shows the frequency vs. time relationships for these trials. The solid line in the middle is the ideal case. Note that as the slope is deviated from the ideal value (1.0), the output pulse flattens out and moves away from the center.



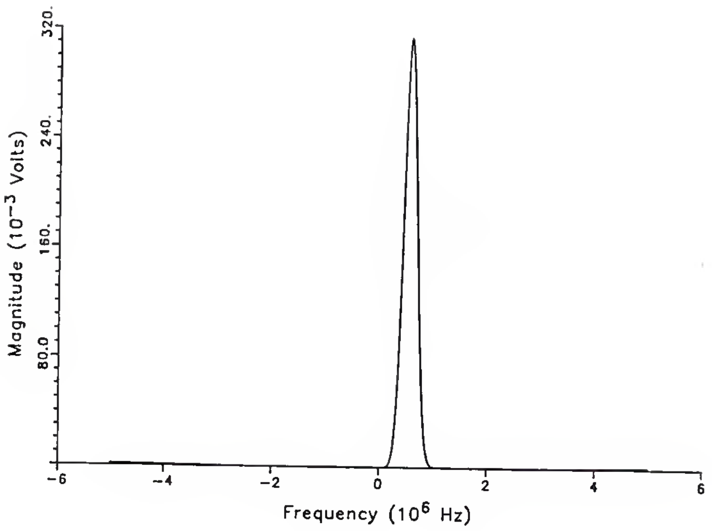
Plot 1A–1 Frequency vs. Time Relationships used for the S.L.O. Linear Errors.



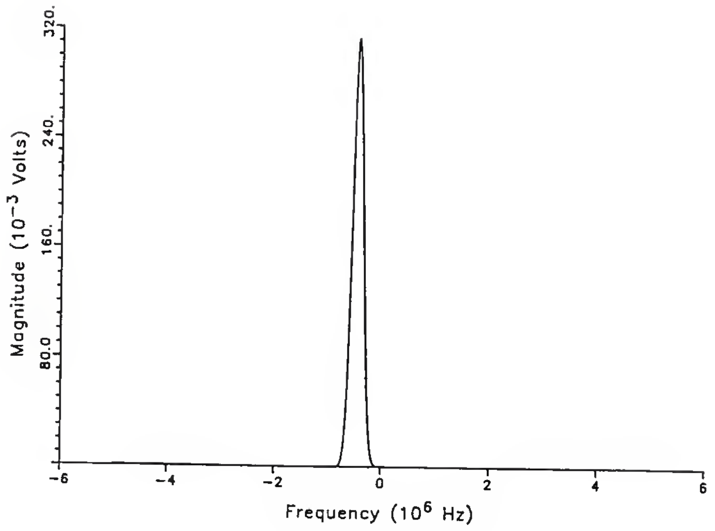
Plot 1A–2 Frequency Compressive Receiver output for S.L.O. Linear Error of 0.5



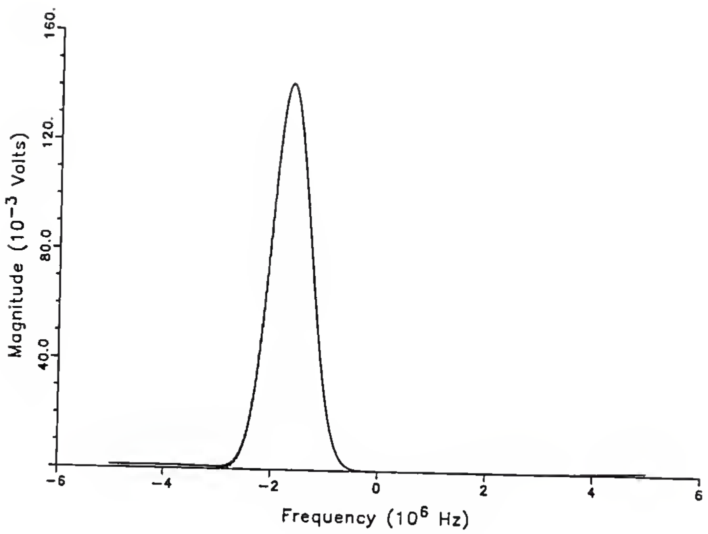
Plot 1A–3 Frequency Compressive Receiver output for S.L.O. Linear Error of 0.7



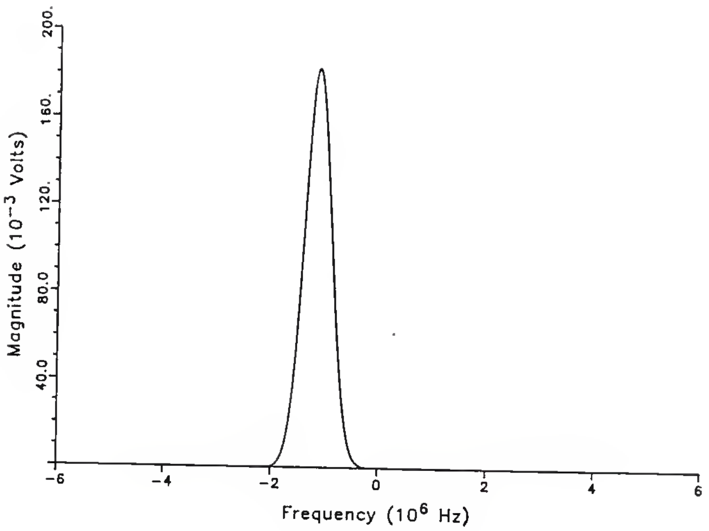
Plot 1A–4 Frequency Compressive Receiver output for S.L.O. Linear Error of 0.9



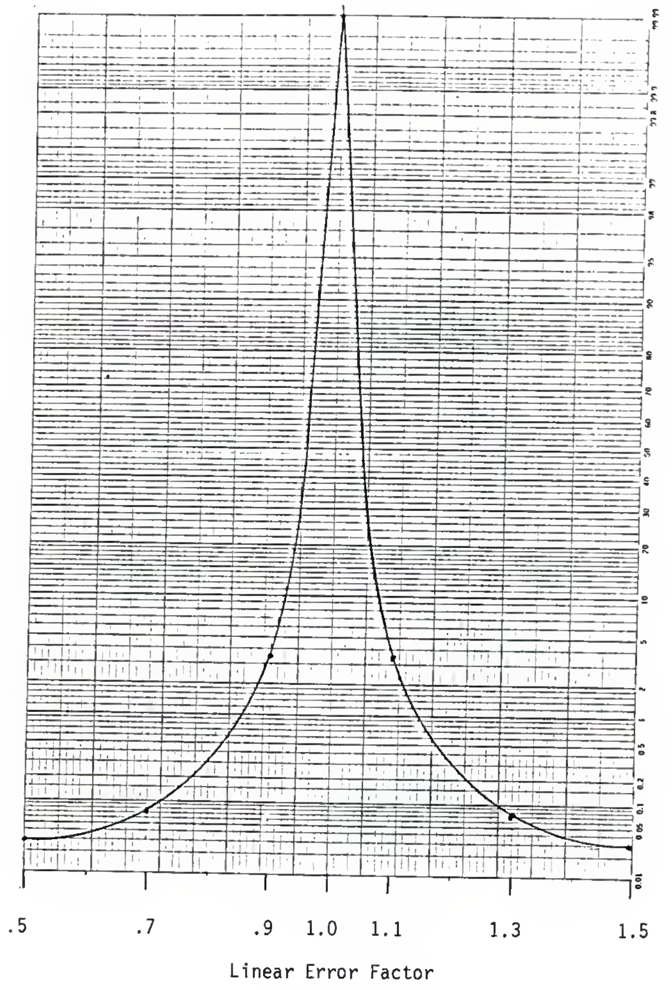
Plot 1A–5 Frequency Compressive Receiver output for S.L.O. Linear Error of 1.1



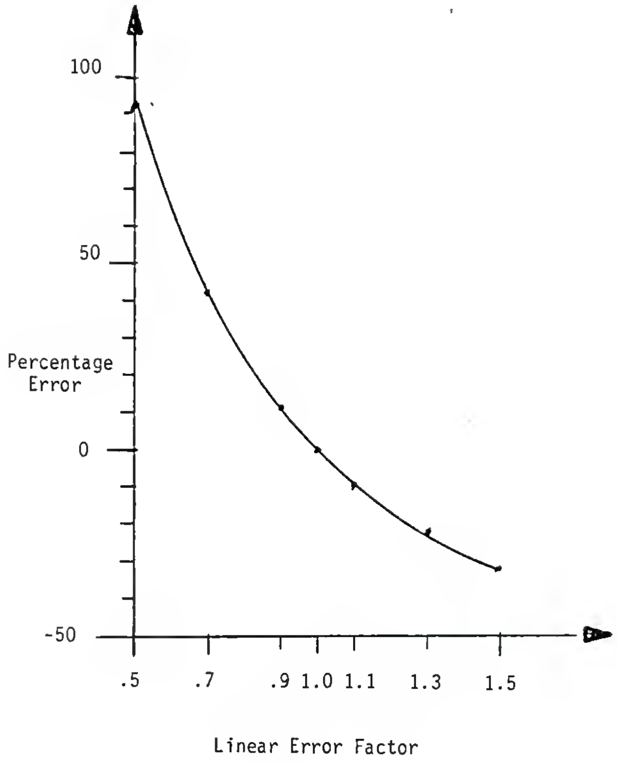
Plot 1A–6 Frequency Compressive Receiver output for S.L.O. Linear Error of 1.3



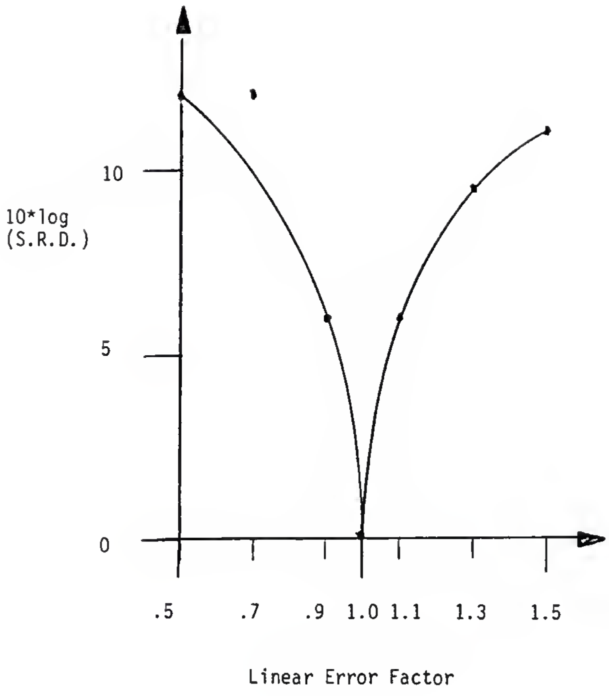
Plot 1A-7 Frequency Compressive Receiver output for S.L.O. Linear Error of 1.5



Graph 1A-1 Probability of Detection vs. S.L.O. Linear Errors.



Graph 1A–2 Frequency Errors vs. S.L.O. Linear Errors



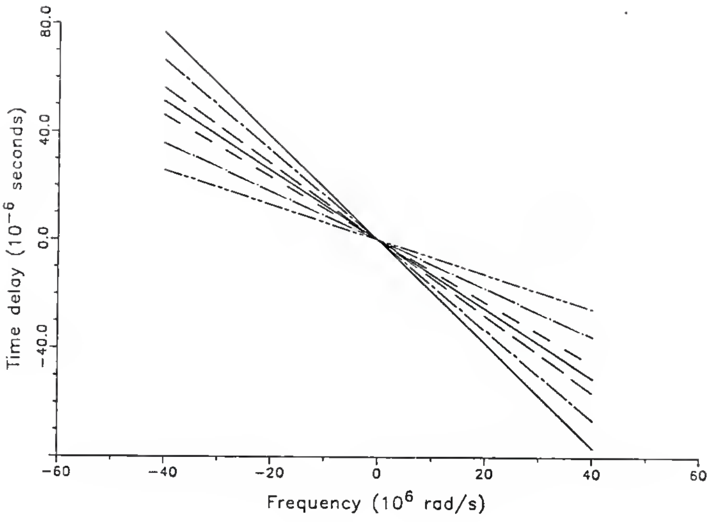
Graph 1A-3 S.R.D. vs. S.L.O. Linear Errors.

Graphs 1A 1–3 show the receiver performance criteria plotted as a function of error. The probability of detection and S.R.D. graphs show a very rapid deterioration as the slope moves away from the ideal. The frequency determination error is assymetrical and roughly linear over part of the range. It appears it is better to overscan than underscan.

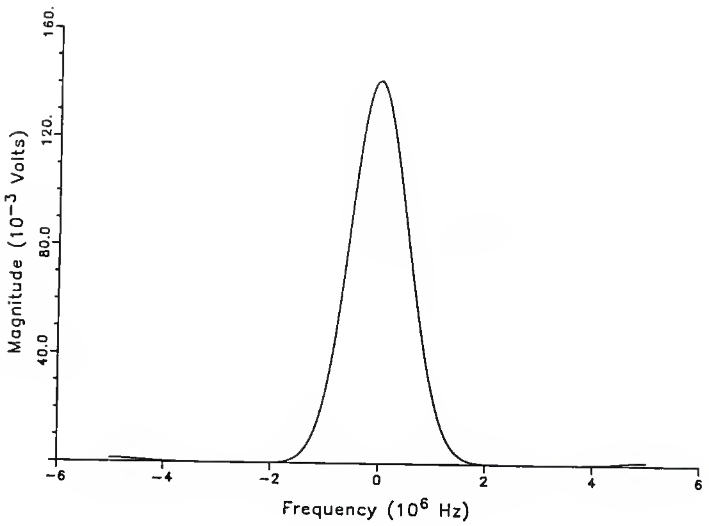
Linear errors in the dispersive filter have the effect of not compressing the spectral components fully. Therefore, the output is not impulsive. The linear error is quantified in the same way as was the error for the S.L.O.

The plots for dispersive filter linear errors are shown in plots 1B 1–6 for error multipliers of 0.50, 0.70, 0.90, 1.10, 1.30 and 1.50 respectively. Plot 1B–7 shows the time delay vs. frequency relations used for these trials, the ideal one being the solid one in the middle. The major difference, with respect to the linear S.L.O. errors, is that no frequency determination errors occurred.

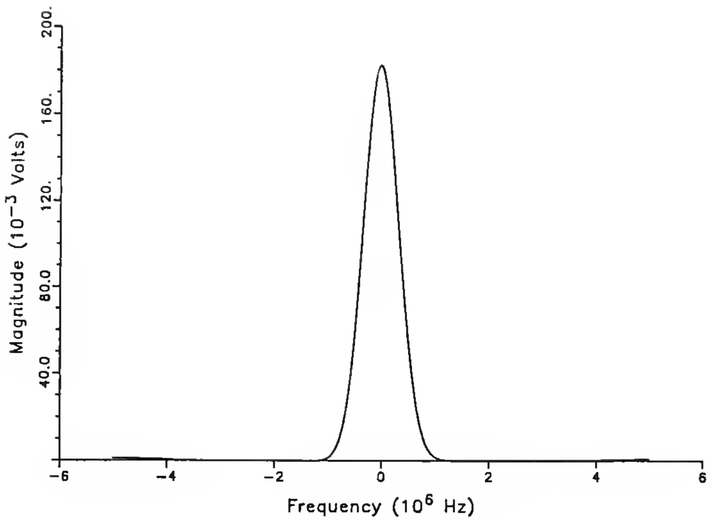
Graphs 1B 1–2 show the receiver performance criteria plotted against the linear error of the dispersive filter. These graphs show approximately the same degradation of performance as did the previous case.



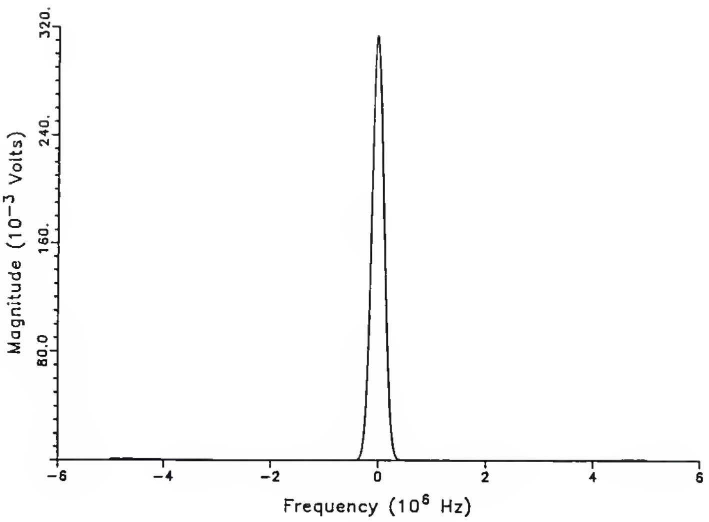
Plot 1B–1 Time Delay vs. Frequency relationships used for the dispersive filter linear errors.



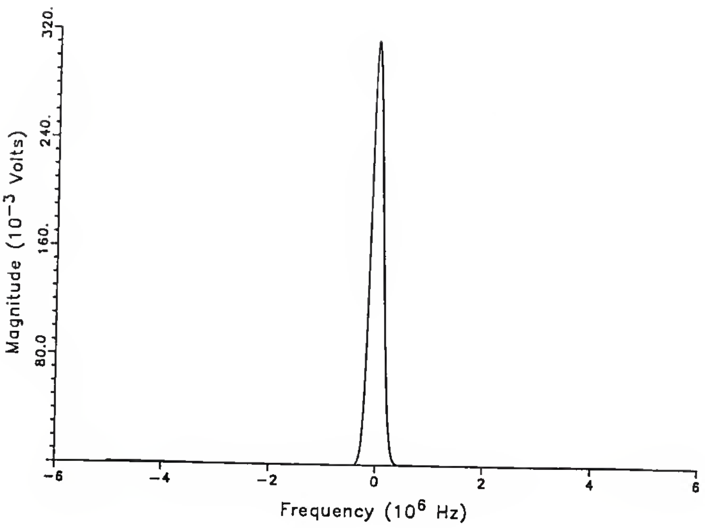
Plot 1B–2 Frequency Compressive Receiver output for dispersive filter linear error of 0.5



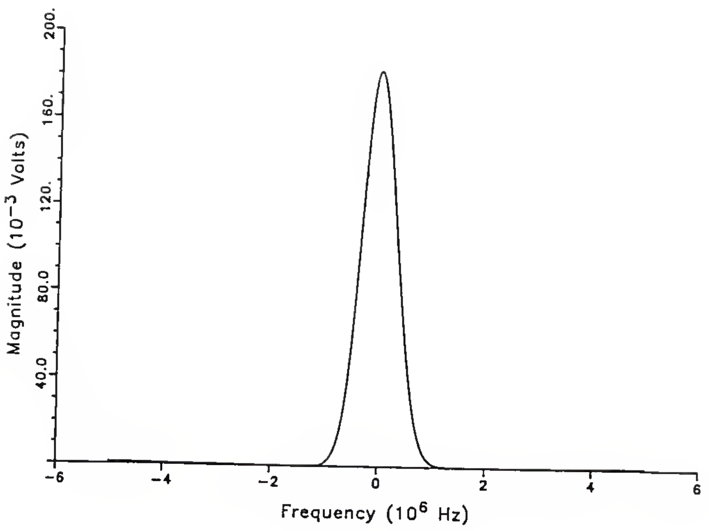
Plot 1B-3 Frequency Compressive Receiver output for dispersive filter linear error of 0.7



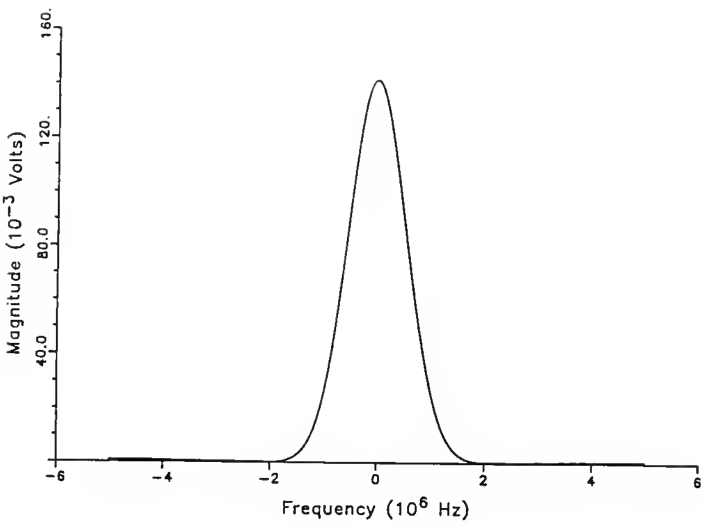
Plot 1B–4 Frequency Compressive Receiver output for dispersive filter linear error of 0.9



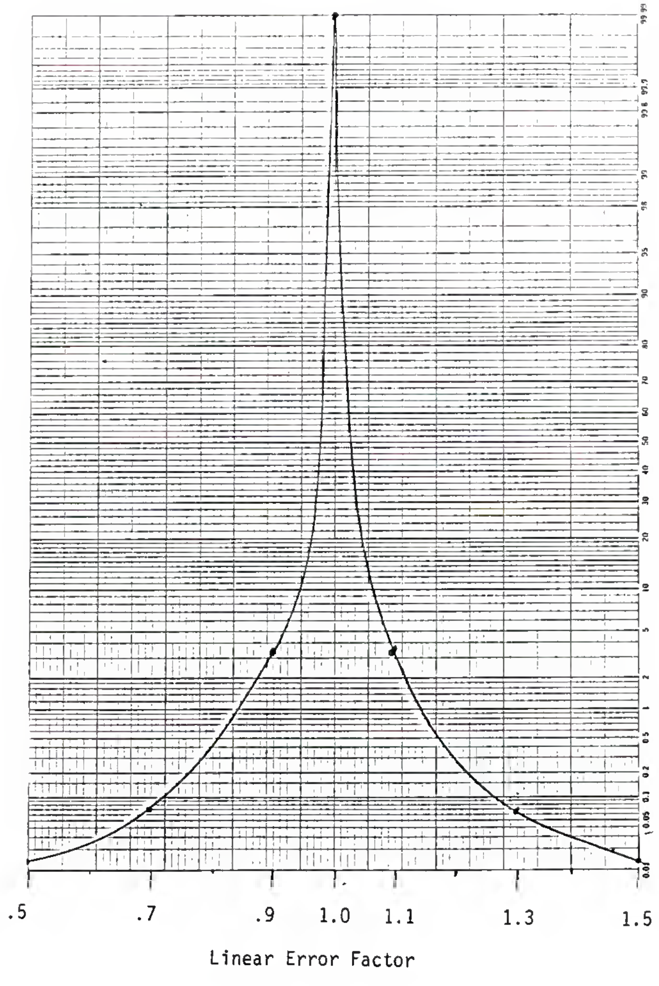
Plot 1B–5 Frequency Compressive Receiver output for dispersive filter linear error of 1.1



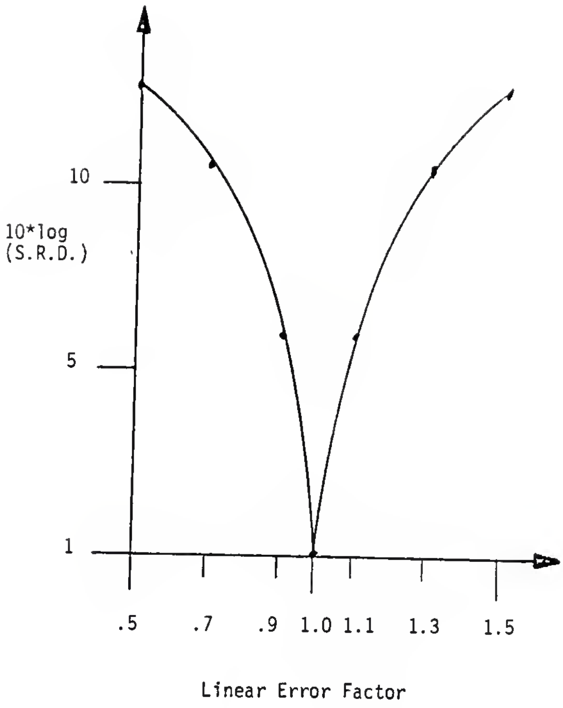
Plot 1B–6 Frequency Compressive Receiver output for dispersive filter linear error of 1.3



Plot 1B–7 Frequency Compressive Receiver output for dispersive filter linear error of 1.5



Graph 1B-1 Probability of detection vs. Dispersive Filter Linear Error.



Graph 1B–2 S.R.D. vs. Dispersive Filter Linear Error.

Section II

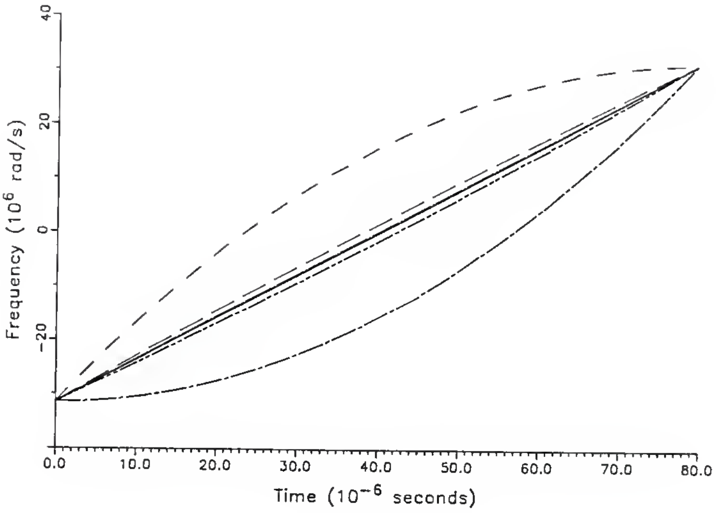
This section is for the non–linear relationships between time and frequency for the S.L.O. and the dispersive filter. The non–linearity used was a quadratic or second–order polynomial. The form was restricted so that there were no cross terms and that the end points of the linear relation were intersected by the polynomial. In other words, the polynomial starts and stops at the ideal values. The final restriction made was that the polynomial had to be monotonic over the range of interest.

Polynomials that are concave up and down over the range were used. Coefficients were found using an interpolating program. Since the two endpoints were known, the value of the polynomial halfway through the range specified it entirely. The range of values used were bounded by the monotonicity restriction on both extremes. The value for the monotonicity restriction was approximately one–half of the end point value. A percentage of this value was used to characterize the degree of offset, or non–linearity. A negative percentage value specifies a concave up curve.

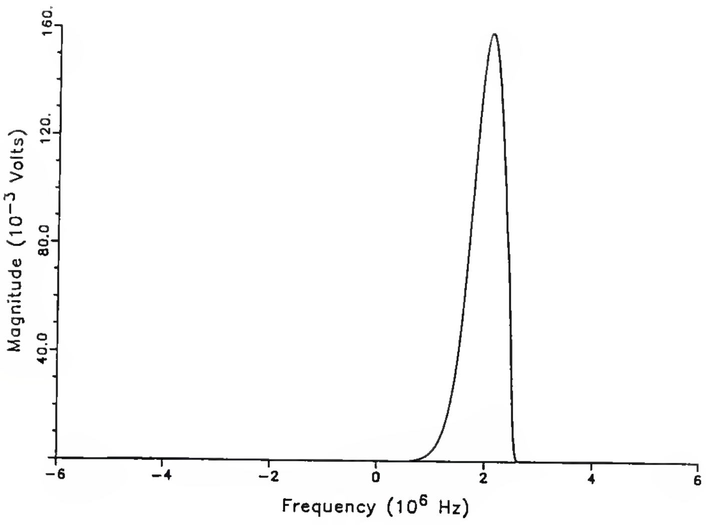
The non–linear error as it has been defined can be thought of as a two–part linear error. For example, the concave up case is scanning too slowly the first half and then scans too rapidly the second half. The same is true of the concave down relationship, just in reverse order.

Plot 2A–1 shows the time delay vs. frequency functions used. The straight solid line in the middle is the ideal case. Plots 2A 2–7 show the output for ± 100, 50, and 10 percent. Performance characteristics are again referenced to the ideal case as before.

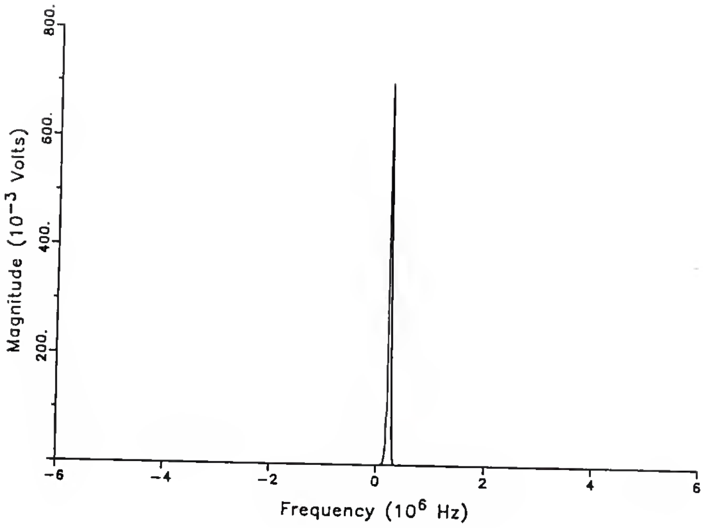
For the non–linear S.L.O., the effect was similar to that of the linear errors, although the output was a little more asymetric. The pulse was



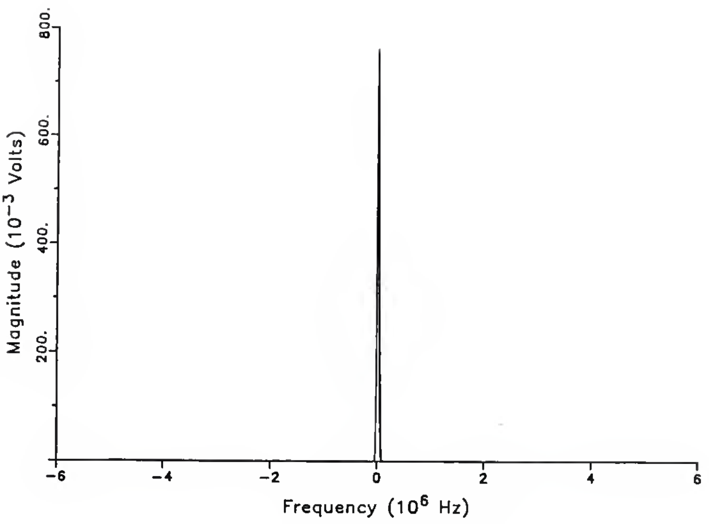
Plot 2A–1 Frequency vs. Time Relationships used for the non–linear S.L.O. errors



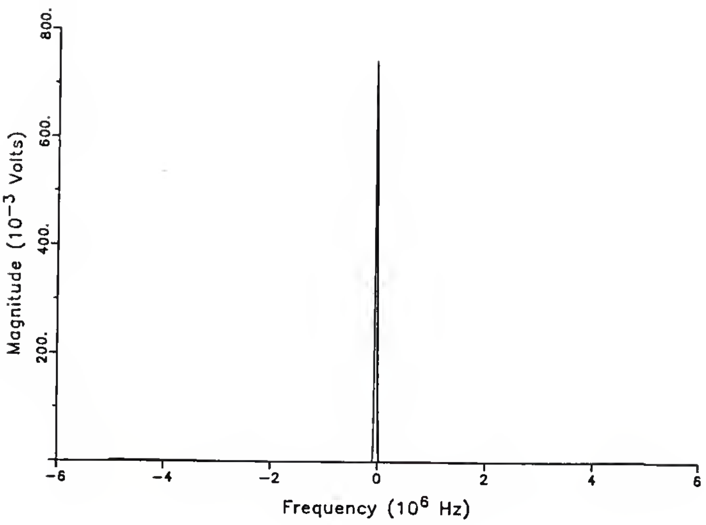
Plot 2A-2 Frequency Compressive Receiver output for S.L.O. non-linear error of - 100%.



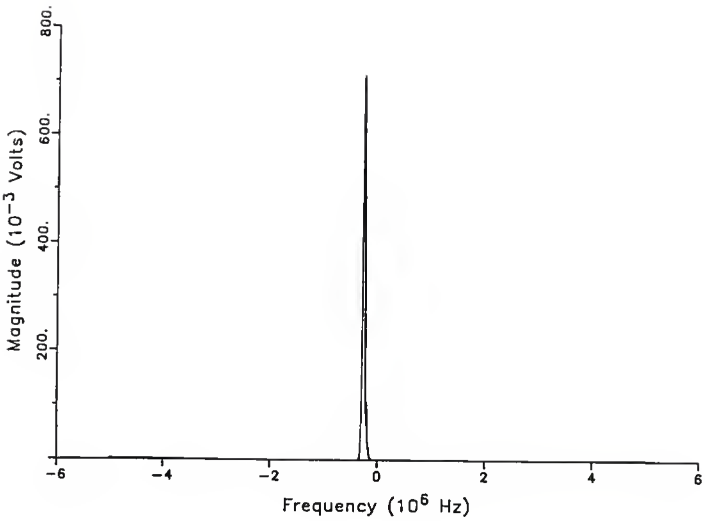
Plot 2A–3 Frequency Compressive Receiver output for S.L.O. non–linear error of – 50%.



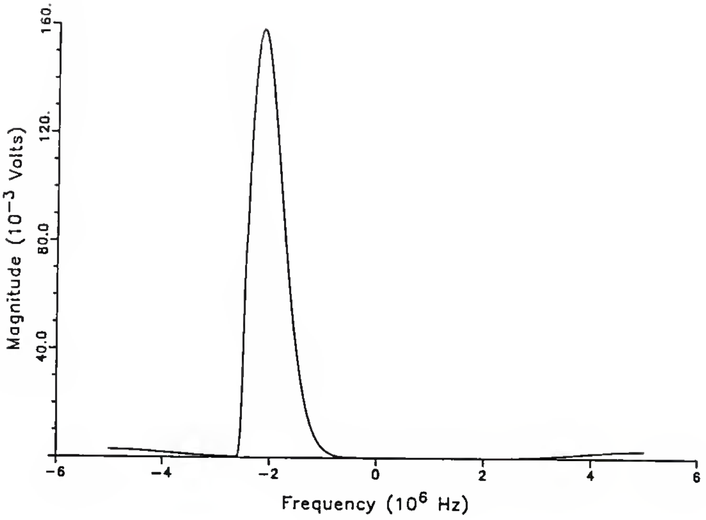
Plot 2A–4 Frequency Compressive Receiver output for S.L.O. non–linear error of – 10%.



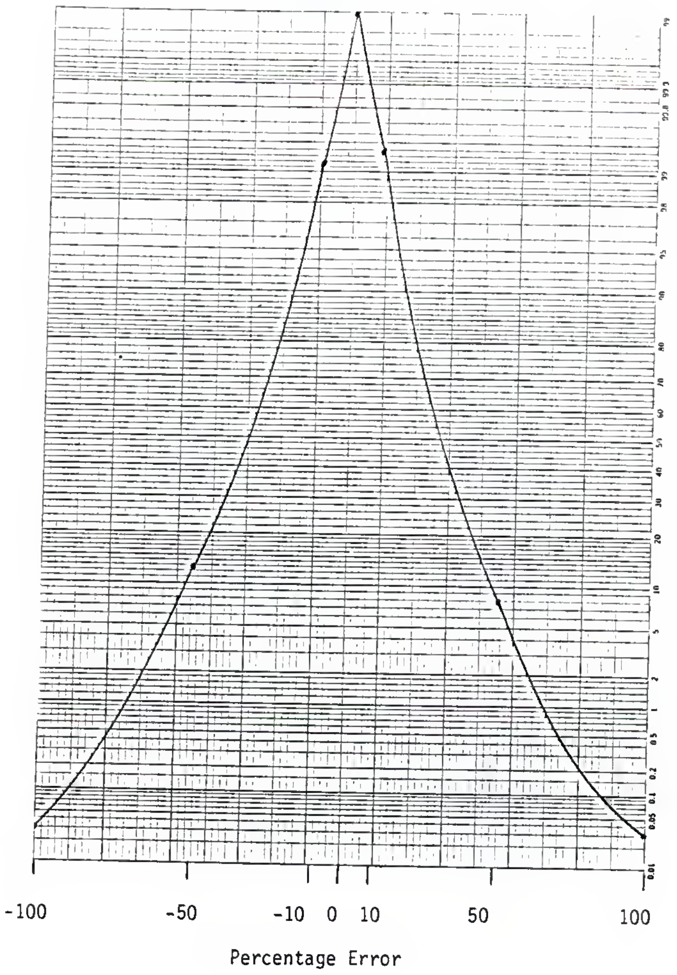
Plot 2A–5 Frequency Compressive Receiver output for S.L.O. non–linear error of 10%.



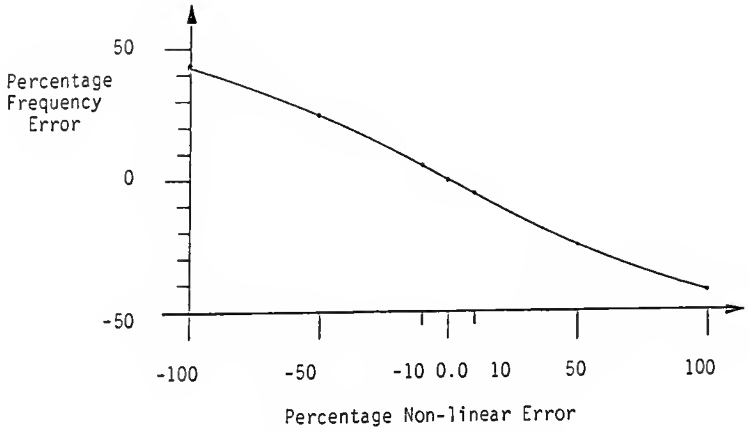
Plot 2A–6 Frequency Compressive Receiver output for S.L.O. non–linear error of 50%.



Plot 2A–7 Frequency Compressive Receiver output for S.L.O. non–linear error of 100%.



Graph 2A–1 Probability of Detection vs. S.L.O. non–linear error.



Graph 2A-2 Frequency Error vs. S.L.O. non-linear error.

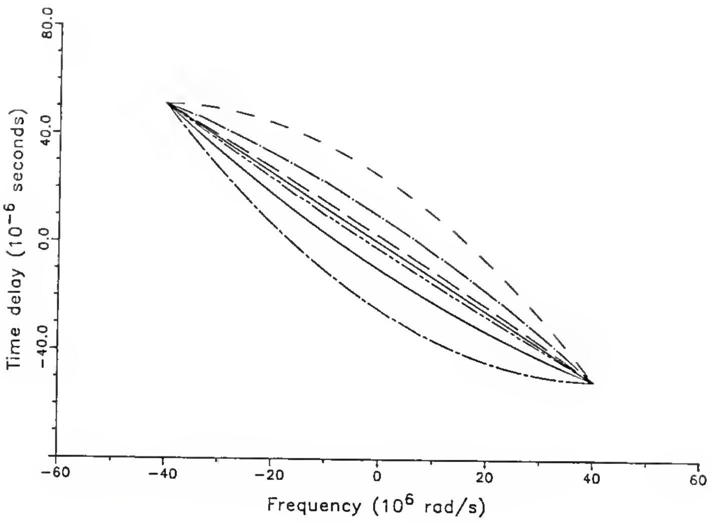
widened, reduced in magnitude, and shifted in frequency with increasing concavity. For the concave up case, the pulse shifted to the right, while concave down shifted it to the left.

Graphs 2A 1–2 show the receiver performance parameters plotted as a function of non–linear error. The curve for probability of detection is much higher for small errors than the linear case, but note that the frequency error is almost a linear function of non–linear error in the way that it has been defined. In regard to SRD, the worst case spreading was a factor of 10.

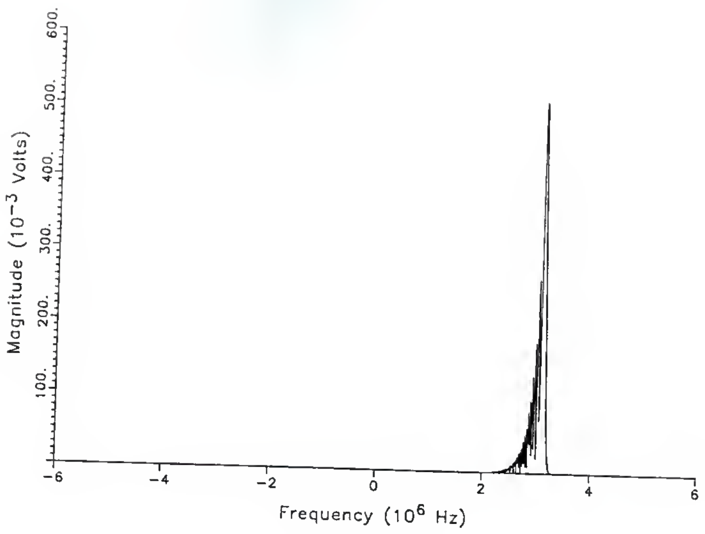
For the non–linear dispersive filter, the same approach was taken as with the S.L.O. Again, about half the value of the endpoint was the limit to maintain monoticity, and the error was defined as a percentage of this value to determine the offset halfway through the scan. Plot 2B–1 shows the non–linear functions used with the solid line in the middle representing the ideal case. Plots 2B 2–7 show the output of the receiver for errors of ± 100, 50, and 10 percent.

Graphs 2B 1–2 show the receiver performance curves. Note that the maximum degradation of peak signal is significantly less than any other case. Again, the frequency determination error is approximately linear with increasing non–linear error as it has been defined. Finally, the maximum spectral resolution degradation was a factor of two.

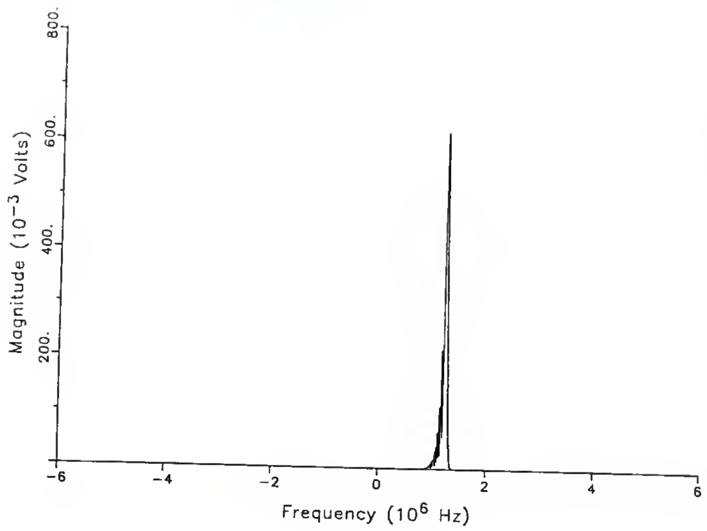
Examining the performance curves, the quadratic dispersive filter performed significantly better than the other three cases in probability of detection. It also had a much smaller maximum S.R.D. and a near linear relation between frequency error and non–linear error in the way that it has been defined.



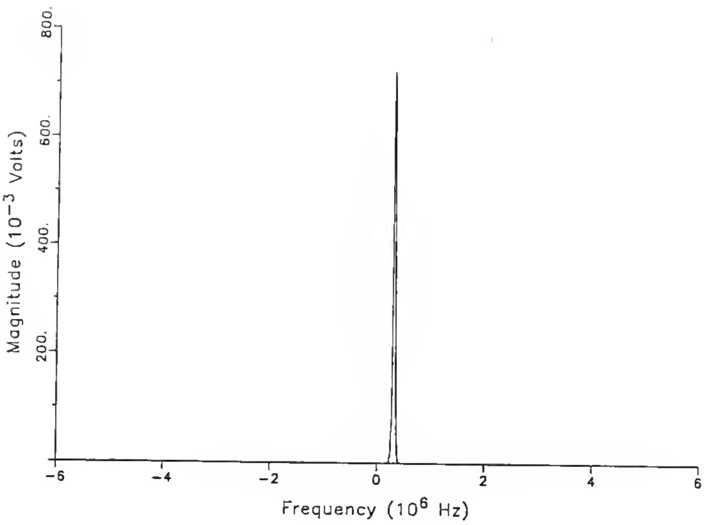
Plot 2B–1 Time Delay vs. Frequency Relations used for the dispersive filter non–linear errors.



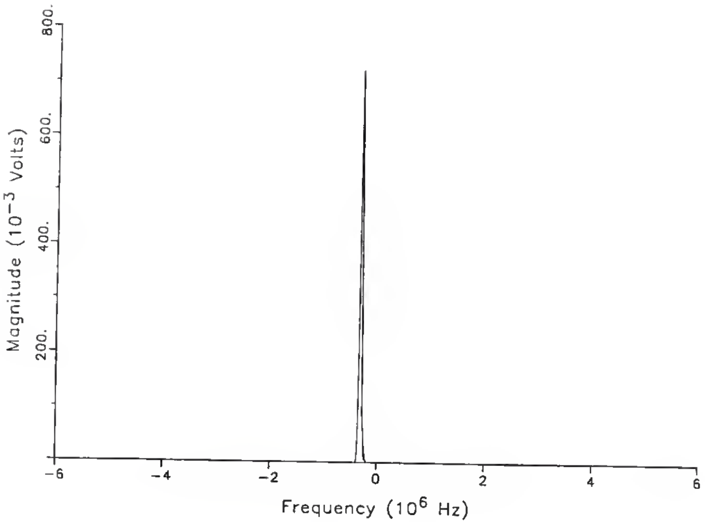
Plot 2B–2 Frequency Compressive Receiver output for dispersive filter non–linear error of − 100%.



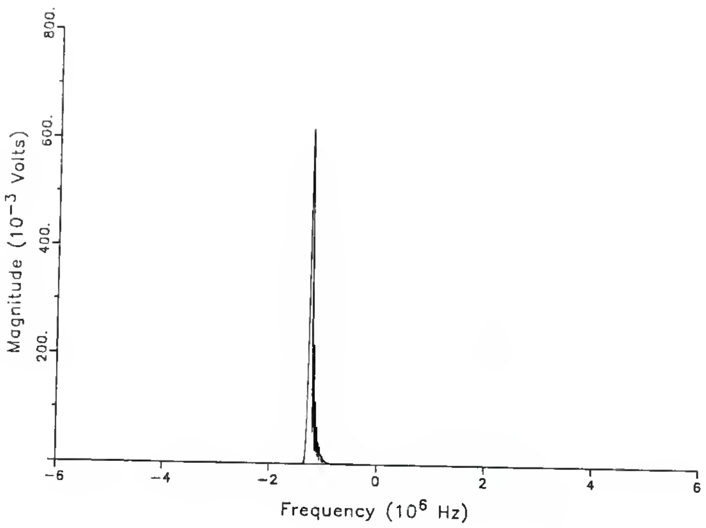
Plot 2B–3 Frequency Compressive Receiver output for dispersive filter non–linear error of – 40%.



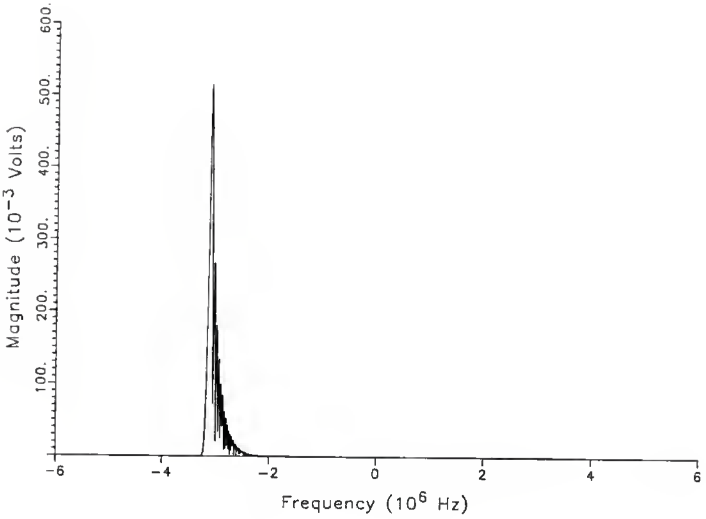
Plot 2B–4 Frequency Compressive Receiver output for dispersive filter non–linear error of – 10%.



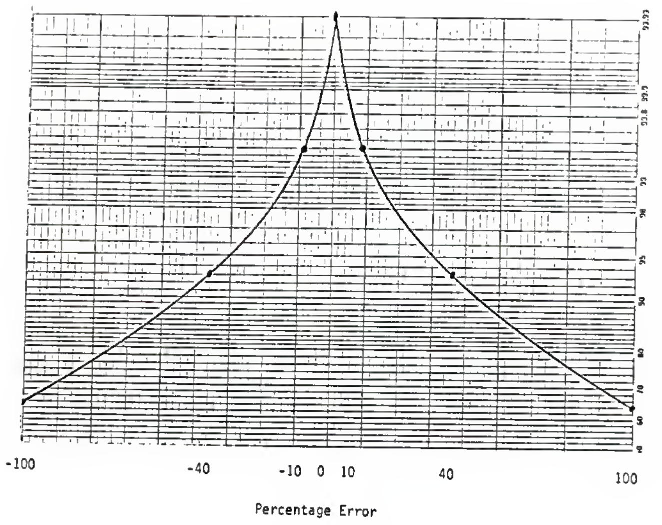
Plot 2B–5 Frequency Compressive Receiver output for dispersive filter non–linear error of 10%.



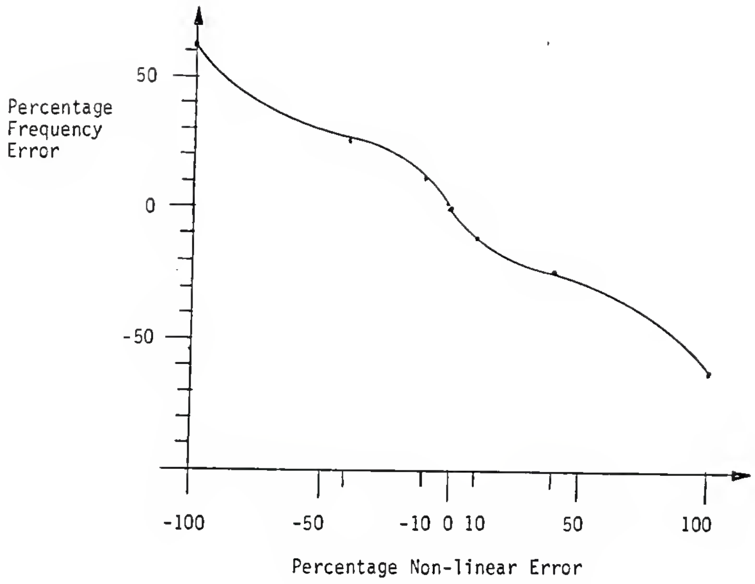
Plot 2B-6 Frequency Compressive Receiver output for dispersive filter non–linear error of 40%.



Plot 2B-7 Frequency Compressive Receiver output for dispersive filter non-linear error of 100%.



Graph 2B-1 Probability of Detection vs. Dispersive Filter non-linear error.



Graph 2B–2 Frequency Errors vs. Dispersive Filter non–linear error.

Section III

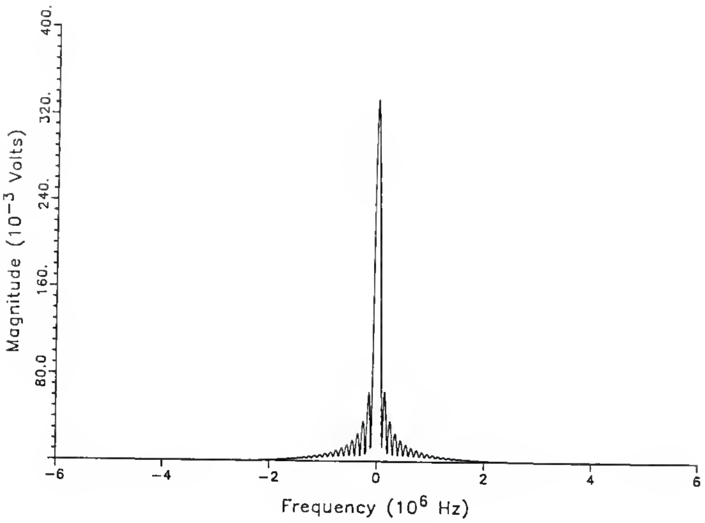
The previous trials were performed using long pulses, which are the ideal input, because the dispersive filter is matched to them. Several trials were run for an ideal S.L.O. and dispersive filter with the pulse less than the dispersion time. Although by convention any pulse that is shorter than twice the dispersion time is a short pulse, a pulse less than the dispersion time demonstrates more clearly the difficulties which arise.

The problem with short pulses is, depending on where in the S.L.O. scan they occur, they can either be partially compressed or not at all. This is because for each frequency, there is a total time equal to the dispersion time, where the mixer terms of the input signal are completely outside the I.F. compressive bandwidth. If the pulse falls completely outside of the compressive I.F. bandwidth, it will not be detected. If part or all of a short pulse is within the compressive bandwidth, the resulting output pulse will have a greatly reduced probability of detection and suffer from assymetries and increased S.R.D.

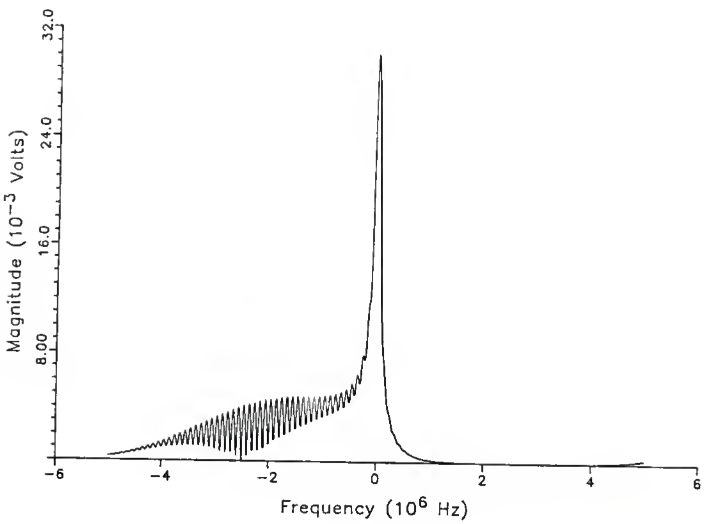
The pulse used was a 10 microsecond rectangular pulse centered in the passband at 0 MHz. The timing of the rising edge of the pulse with respect to the beginning of the scan time was adjusted, so the pulse fell in different places in the I.F. compressive bandwidth. The positions were: dead center, half in and half out on both edges, and between the two previous positions on both sides. These positions correspond to time delaying the pulse 35, 15, 55, 25, and 45 microseconds respectively. With the I.F. compressive bandwidth timing window 40 microseconds wide, even if all the input pulses I.F. terms fall in the bandwidth, it only occupies one fourth the space. This results in a reduction in peak magnitude.

Plots 3A 1–5 show the output pulses for the above cases. Note how wide the output pulses are and how slowly they fall off. The best case probability of detection for this particular input was 5.6 percent for the pulse delayed 35 microseconds.

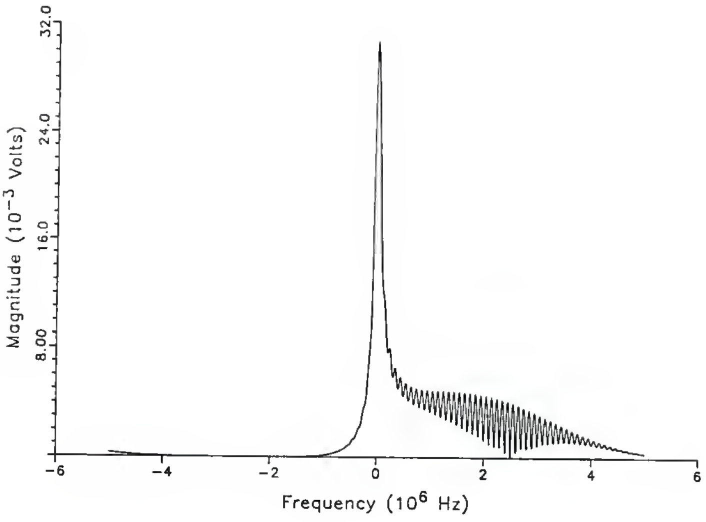
Plots 3B 1–5 show the same inputs, but with a linear S.L.O. error of 0.9 percent. Now an additional problem appears in that a frequency determination error appears. In short, short pulses are difficult to receive.



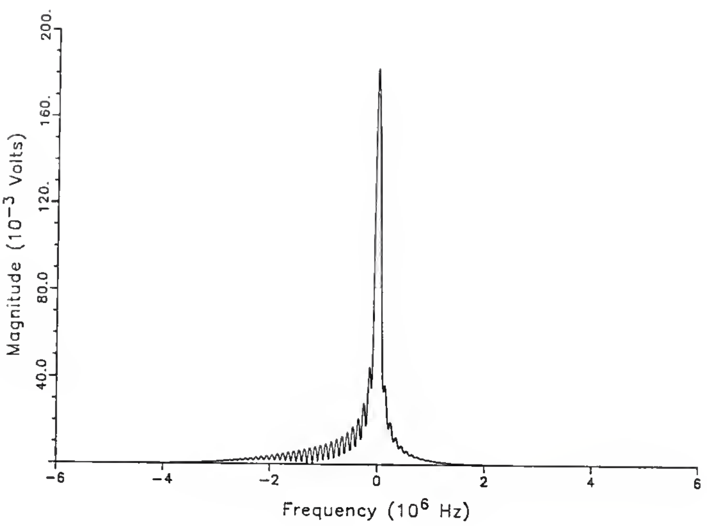
Plot 3A–1 Frequency Compressive Receiver output for a short pulse delayed 35 microseconds.



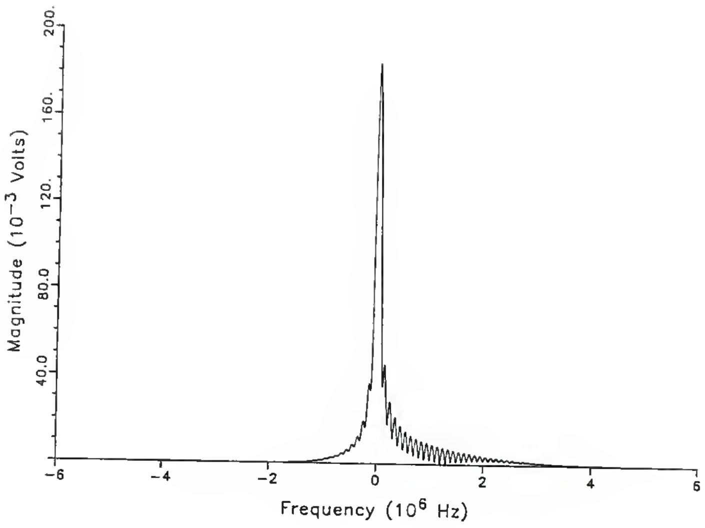
Plot 3A–2 Frequency Compressive Receiver output for a short pulse delayed 15 microseconds.



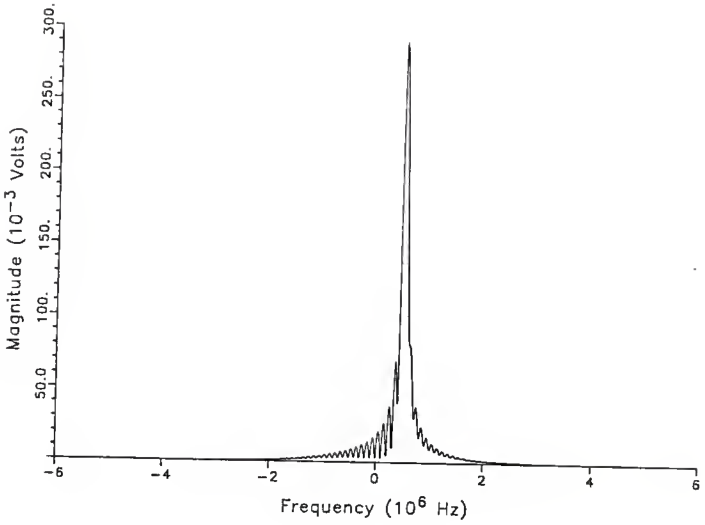
Plot 3A-3 Frequency Compressive Receiver output for a short pulse delayed 55 microseconds.



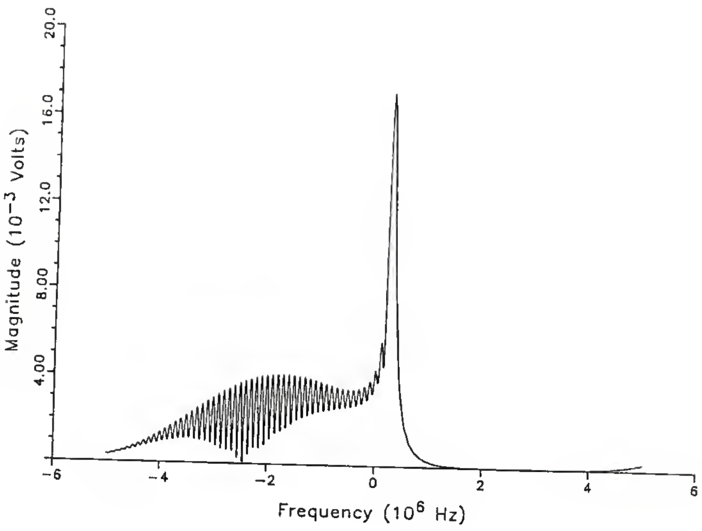
Plot 3A–4 Frequency Compressive Receiver output for a short pulse delayed 25 microseconds.



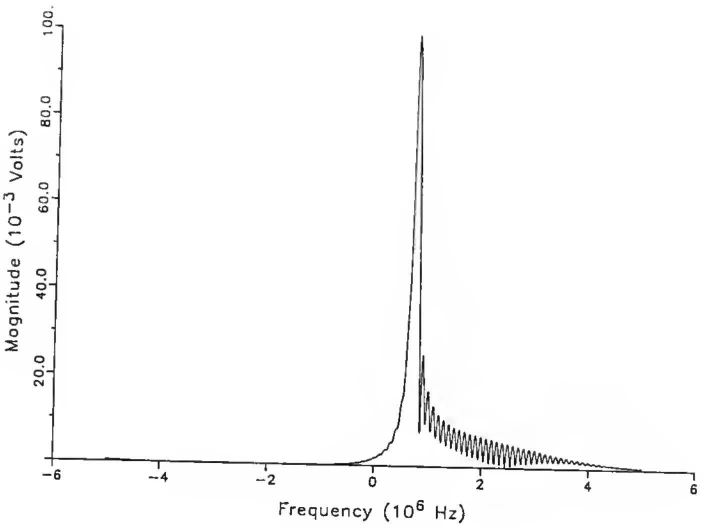
Plot 3A-5 Frequency Compressive Receiver output for a short pulse delayed 45 microseconds.



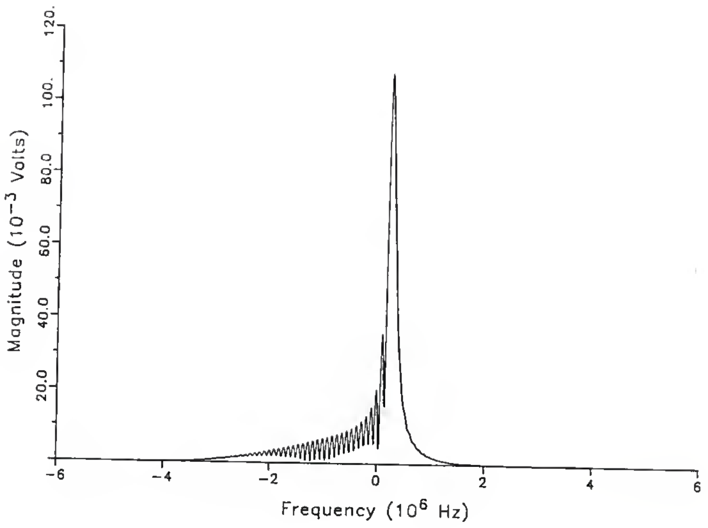
Plot 3B-1 Output for a Frequency Compressive Receiver with a S.L.O. linear error of 0.9 for a short pulse delayed 35 microseconds.



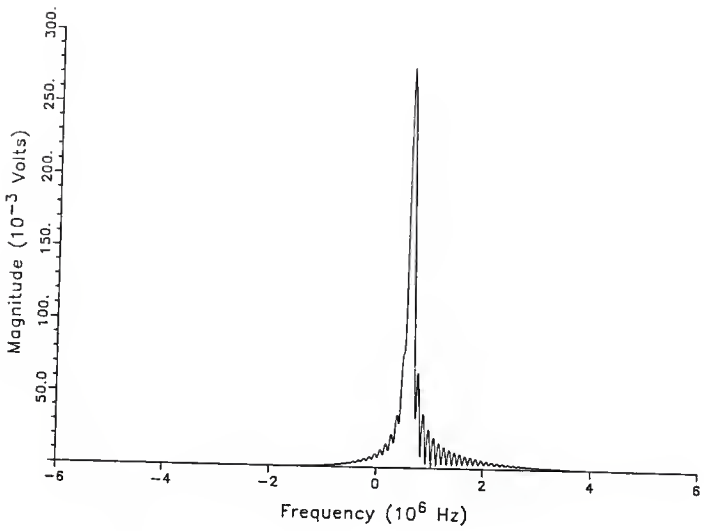
Plot 3B–2 Output for a Frequency Compressive Receiver with a S.L.O. linear error of 0.9 for a short pulse delayed 15 microseconds.



Plot 3B-3 Output for a Frequency Compressive Receiver with a S.L.O. linear error of 0.9 for a short pulse delayed 55 microseconds.



Plot 3B–4 Output for a Frequency Compressive Receiver with a S.L.O. linear error of 0.9 for a short pulse delayed 25 microseconds.



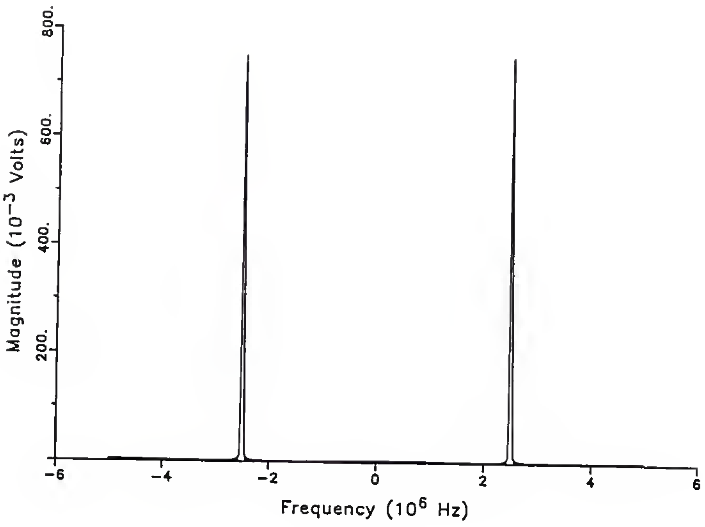
Plot 3B–5 Output for a Frequency Compressive Receiver with a S.L.O. linear error of 0.9 for a short pulse delayed 45 microseconds.

Section IV

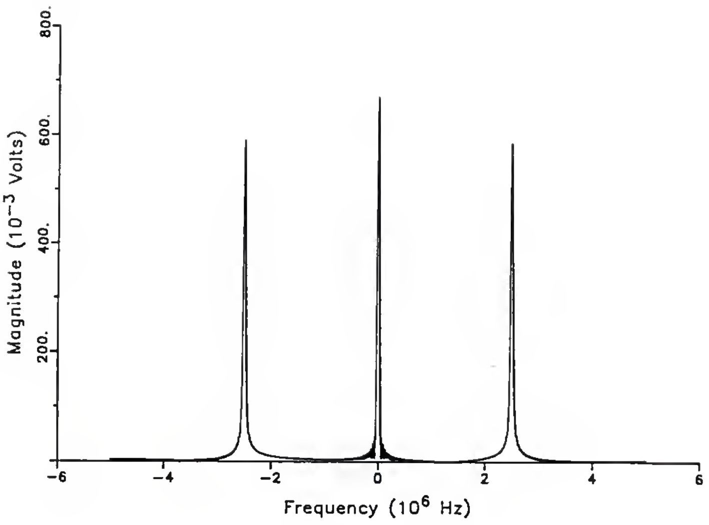
This section describes simulation results for long FSK (Frequency Shift Keying) pulses as inputs to an ideal frequency compressive receiver. No short pulses were simulated, although the capability exists. The performance criteria examined were probability of detection and frequency determination.

The FSK signal was generated by dividing the input pulse into N regions of equal duration. Each region was then modulated by the desired carrier frequency. One would expect the different frequencies of the input signal to be separated by the dispersive filter and appear as distinct pulses on the output. Plot 4–1 does indeed show two distinct output pulses for the two input frequencies of –2.5 and 2.5 MHz. Note that the output peaks are of the same magnitude as a simple long CW input in plot 1.

The second plot is for frequencies of –2.5, 0, and 2.5 MHz respectively. Again, plot 4–2 show the expected spectral sorting. However, now the magnitudes of the output pulses are not the same. Although each carrier frequency receives an equal amount of time in the input, the middle frequency has the highest peak. This can be explained by recalling that for full compression, an input pulse for a given frequency must be present for a period of time τ over a specific range of the sweep in order for all of its IF components to fall within the dispersive filter's IF bandwidth. For an input pulse of a frequency at the lower compressive band edge, this time is from 0 to τ. The time for a pulse with a frequency on the upper band edge is from τ to 2 τ. If the scan time were only divided for these two frequencies, then both would experience full compression as in plot 4–1. However, the third pulse at the center frequency upsets this. For the center frequency to be fully compressed, it must be present from 1/2 τ to 3/2 τ. Assuming the scan rate



Plot 4–1 Frequency Compressive Receiver output for an FSK input of –2.5 and 2.5 MHz.

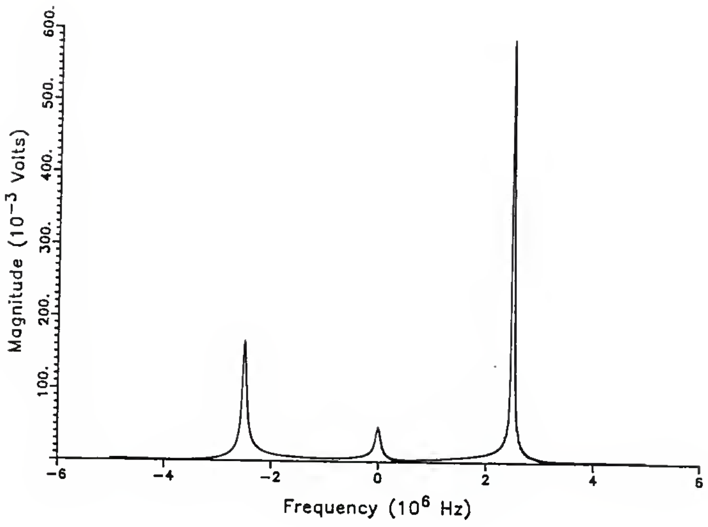


Plot 4–2 Frequency Compressive Receiver output for an FSK input of –2.5, 0.0, and 2.5 MHz.

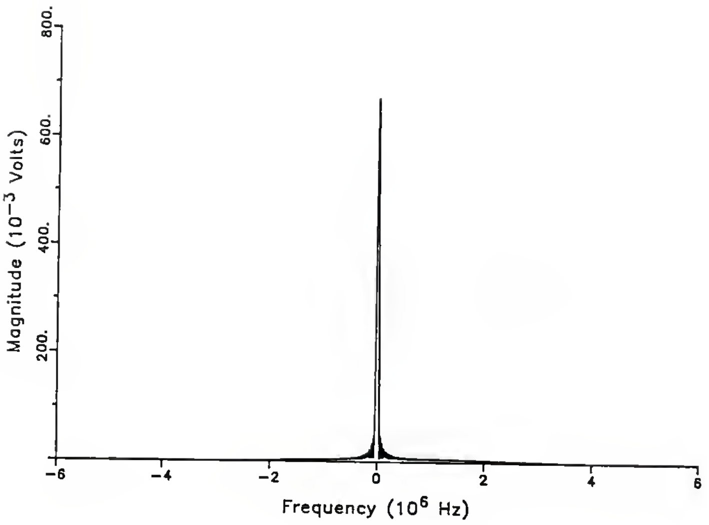
is divided equally into three regions, more of the center frequency's IF terms fall within the IF bandwidth of the dispersive filter, at the expense of the two adjacent frequencies. This accounts for the disparity in output pulse peaks.

Plots 4–3 and 4–4 further illustrate the problem. An input signal with three frequency shifts is the input to the receiver as in plot 4–2, only the frequency order is changed. Plot 4–3 shows the output for a frequency order of 0, –2.5, and 2.5 MHz. Note the reduction of the peaks for –2.5 and 0 MHz. This is because very little of the IF terms for these frequencies fall within the IF bandwidth. Finally, plot 4–4 shows the output for a frequency order of 2.5, 0, and –2.5 MHz. The ± 2.5 MHz signals are undetectable because none of their IF terms fell within the IF bandwidth of the compressive filter.

In order to determine all the frequencies used in an FSK signal, one sweep is inadequate. First assume the receiver has adequate spectral resolution to pick out the different frequency components and that each frequency used has an equal probability of occuring. Depending on the number of frequency transitions per sweep and the order the frequencies occur in with respect to the sweep, some frequencies are less likely to be detected than others for a given sweep. Detection of some frequencies for a given sweep may be impossible, as shown in plot 4–4. Therefore, multiple sweeps of an FSK signal are needed to insure that all frequencies used are detected.



Plot 4–3 Frequency Compressive Receiver output for an FSK input of 0.0, –2.5 and 2.5 MHz.



Plot 4–4 Frequency Compressive Receiver output for an FSK input of 2.5, 0.0 and –2.5 MHz.

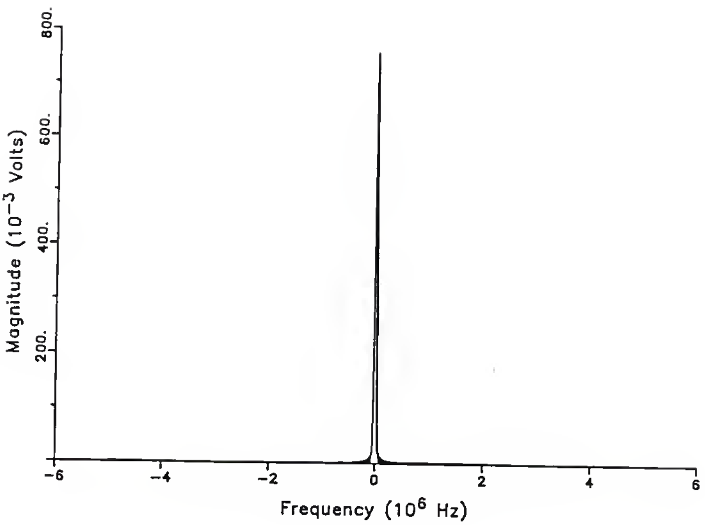
Section V

This section describes the simulation results for long PSK (Phase Shift Keying) pulse inputs to an ideal frequency compressive receiver. No short pulses were simulated, although the capability existed. The only performance criterion reported was the probability of detection which was referenced to the ideal output pulse shown in plot 1.

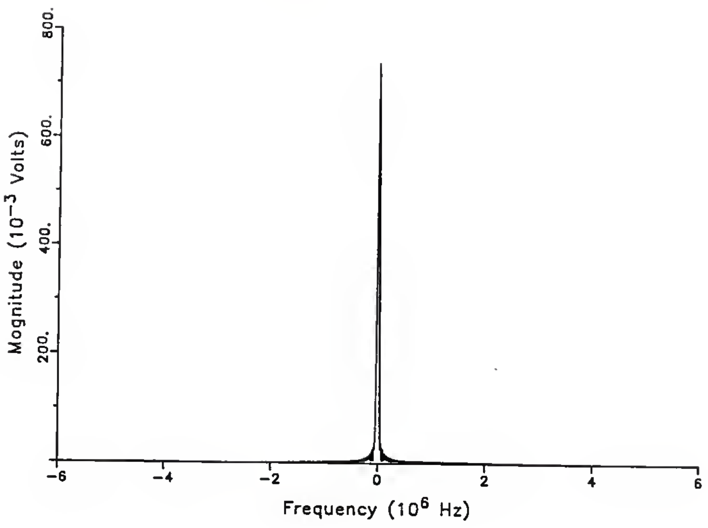
The PSK pulse was generated by dividing equally the scan time by the desired number of phase values. The phase of the region was superimposed by setting the baseband signal's complex components to the corresponding trignometric ratio. Although the phase could be set to any arbitrary value, it was alternated between zero and the phase shift under simulation. For example, BPSK (Bi PSK) values were 0, 180, 0, 180 ··· degrees. 8–PSK phase values were 0, 45, 0, 45 ··· degrees.

The first group of plots shows how probability of detection was affected by PSK inputs. The input PSK pulse had three phase values at center band. Plots 5–1 through 5–4 show the receiver output for phase shifts of 22.5, 45, 90 and 180 degrees, corresponding to 16–PSK, 8–PSK, QPSK, and BPSK respectively. Graph 5–1 shows probability of detection vs. phase shift. Note that as the amount of phase shift increased, the peak of the output pulse fell and its base became broader. As the phase shift becomes smaller, the output pulse approached the ideal output. The effect of varying the phase was to broaden the signal's spectrum.

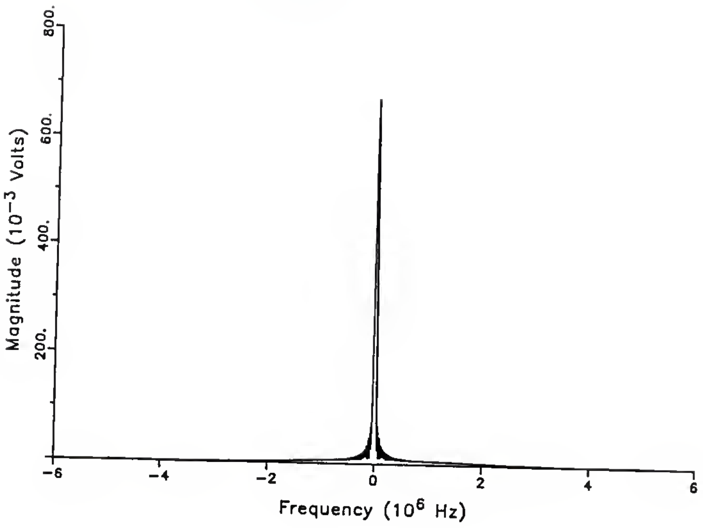
The next group of plots are similar to the previous ones, except that now two phase transitions occur over the scan time. Plots 5–5 through 5–8 show the resulting frequency compressive receiver output for phase shifts of 16–PSK, 8–PSK, QPSK, and BPSK respectively. Graph 5–2 shows the



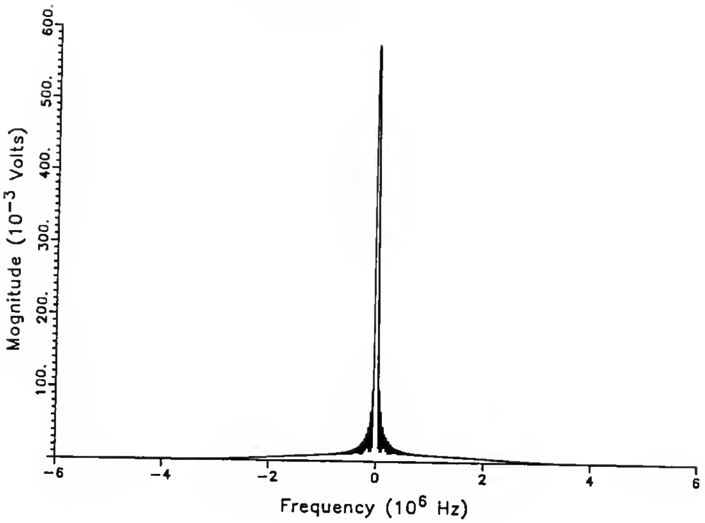
Plot 5–1 Frequency Compressive Receiver output for a three transition 16PSK input.



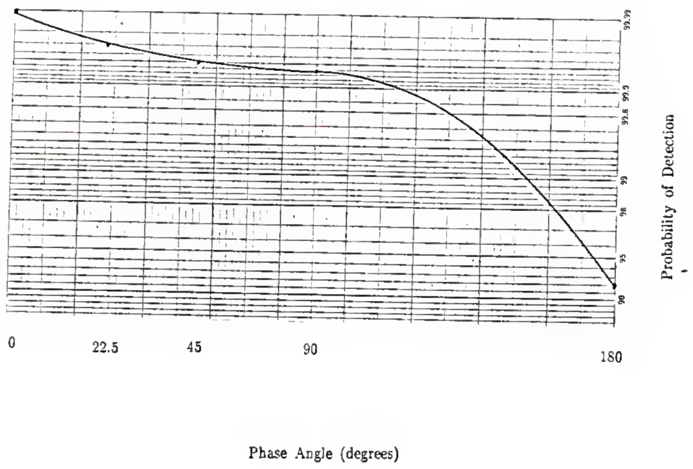
Plot 5–2 Frequency Compressive Receiver output for a three transition 8PSK input.



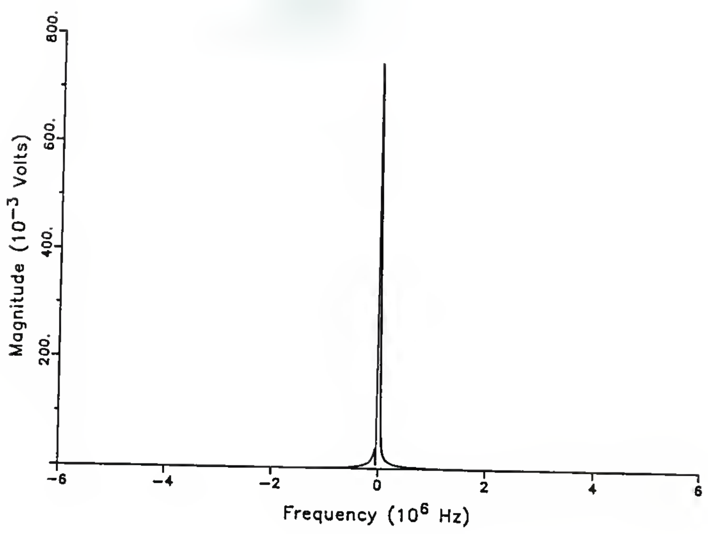
Plot 5–3 Frequency Compressive Receiver output for a three transition QPSK input.



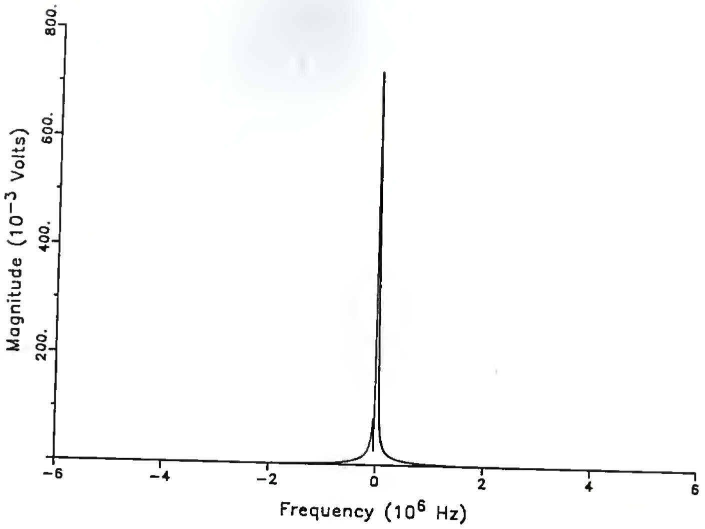
Plot 5–4 Frequency Compressive Receiver output for a three transition BPSK input.



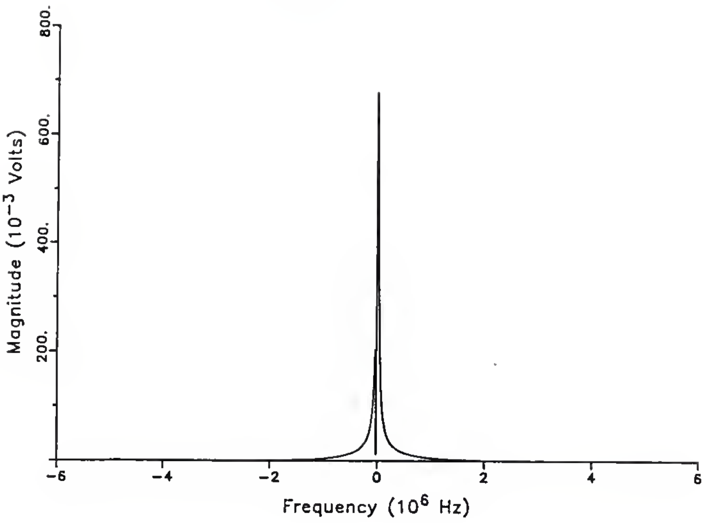
Graph 5–1 Probability of Detection vs. Phase Shift for three transition PSK.



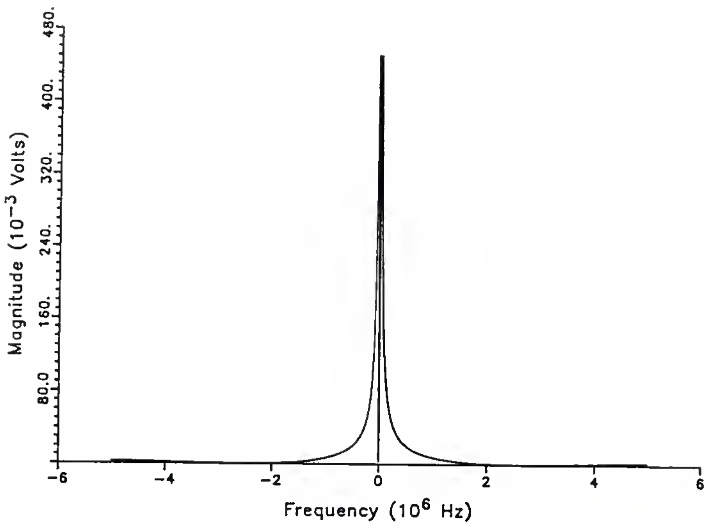
Plot 5–5 Frequency Compressive Receiver output for a two transition 16PSK input.



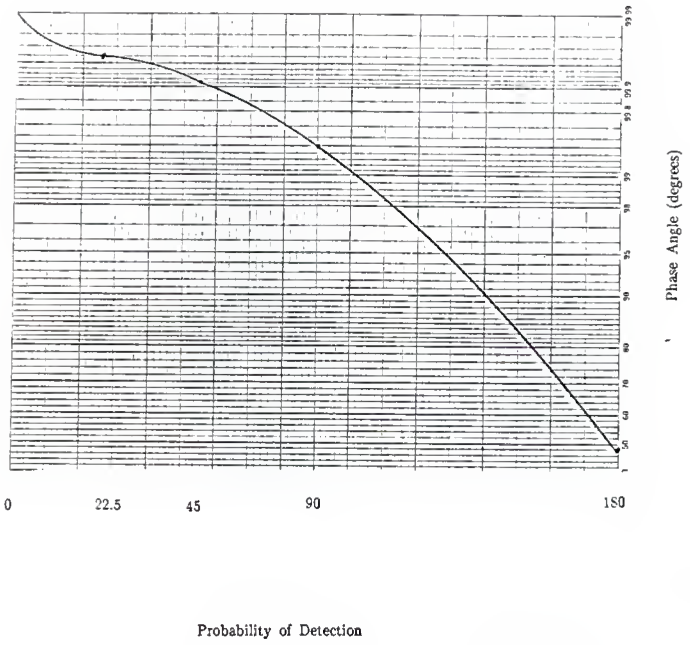
Plot 5–6 Frequency Compressive Receiver output for a two transition 8PSK input.



Plot 5–7 Frequency Compressive Receiver output for a two transition QPSK input.



Plot 5–8 Frequency Compressive Receiver output for a two transition BPSK input.



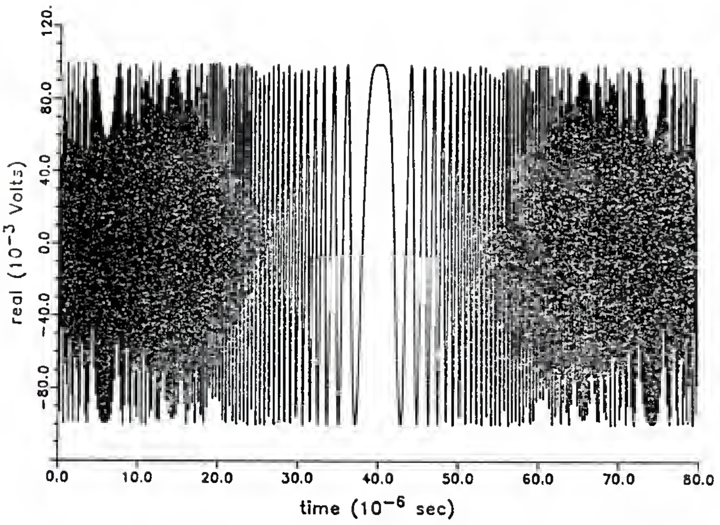
Graph 5–2 Probability of Detection vs. Phase Shift for two transition PSK.

probability of detection vs. phase shift. These plots are different from the previous group in that a notch appeared in the output pulse and increased in depth with increasing phase shift. At 180 degrees, the notch completely divided the pulse in half.

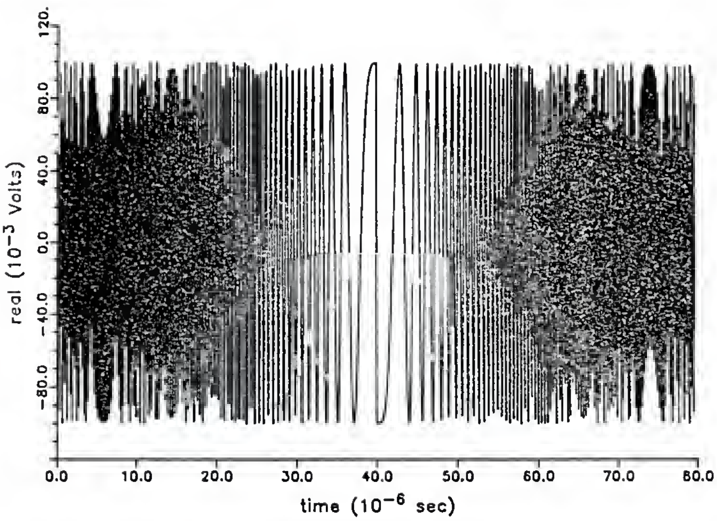
By experimenting with the simulation, it was determined that a notch appears whenever the frequency of the input signal corresponds to the position of the phase transition. That is, the position on the input time domain axis where the phase transition occured corresponds to the same position on the time domain scale on the output, which is calibrated in frequency as a function of time. For example, if the input frequency of the previous group of plots had been ± 1.67 MHz, a similar notch would have appeared.

The notch may be explained analytically in the following way. First, the dispersive filter is broken into a cascade of two sections. The first section was an ideal dispersive filter with a flat magnitude response. The second section is the Gaussian magnitude response with no phase shift. Now recall that for the low pass model, both positive and negative frequencies are used. For an ideal long CW pulse with no phase shift, the time domain output of the mixer is shown in plot 5–9. The positive frequencies on the left hand side are delayed and the negative frequencies on the right hand side are accelerated. Thus, the energy of the positive frequencies comes out in an impulse as does the energy of the negative frequencies. These two impulses combine to form a single large impulse. This impulse then shocks the magnitude response filter and its impulse response is the receiver output.

Now consider the case when the long CW pulse undergoes an 180 degree phase shift half way through the S.L.O. sweep. The mixer output is the same as before for the first half of the sweep, but now plot 5–10 shows how the



Plot 5-9 Mixer Output in the time domain for a constant phase CW input.



Plot 5-10 Mixer Output in the time domain for a CW input with a 180 degree phase shift half way through the sweep.

output for the second half of the sweep is inverted from the previous plot. The positive and negative frequencies are delayed as before, but now the impulses generated by the positive and negative frequencies are of opposite sign when they meet in the middle. In network analysis, this is called a doublet.

The resulting output for when a filter is shocked by doublet is shown analytically here. The output of the magnitude response filter section to the doublet input is given by equation (19),

$$y(t) = g(t) * h(t) \tag{19}$$

where y(t) is the filter output, h(t) is the filter's time domain response and g(t) is the doublet input. The doublet input, g(t), can be represented graphically as in figure 4 when ϵ goes to zero. Convolving g(t) and h(t) yields equation (20). Graphically, this appears as figure 5.

$$y(t) = \int_{-\infty}^{+\infty} h(\tau)\, g(t-\tau)\, d\tau \tag{20}$$

Equation (20) can now be rewritten as

$$y(t) = \frac{-1}{\epsilon^2} \int_{t-\epsilon}^{t} h(\tau)\, d\tau + \frac{1}{\epsilon^2} \int_{t}^{t+\epsilon} h(\tau)\, d\tau\,. \tag{21}$$

As ϵ approaches zero, the integration may be approximated as in figure 6. Now, y(t) in equation (21) can be approximated as equation (22).

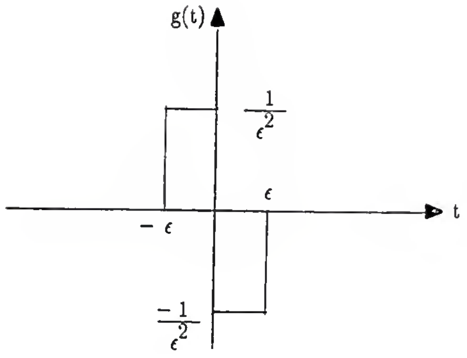


Figure 4 The doublet, g(t), as a limit.

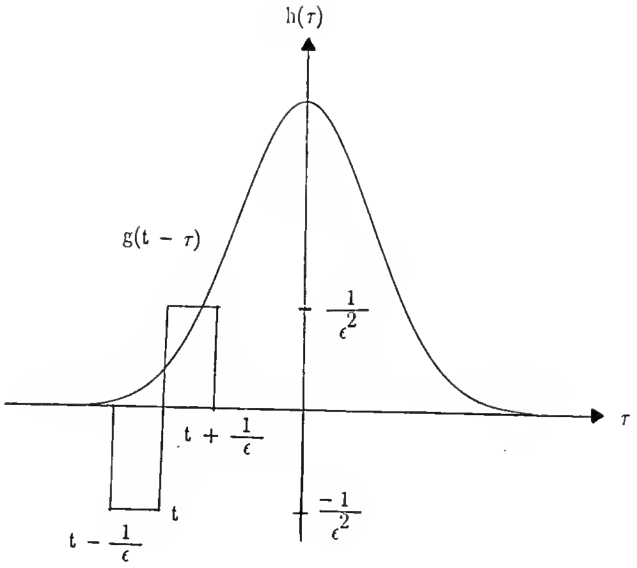


Figure 5 Convolution of h(t) and g(t).

$$y(t) \approx \frac{-1}{\epsilon^2} h\left[t - \frac{\epsilon}{2} \right] \epsilon + \frac{1}{\epsilon^2} h\left[t + \frac{\epsilon}{2} \right] \epsilon \qquad (22)$$

Taking the limit as ϵ goes to zero yields

$$\lim_{\epsilon \to 0} y(t) = \frac{h\left[t + \frac{\epsilon}{2} \right] + h\left[t - \frac{\epsilon}{2} \right]}{\epsilon} = h'(t) . \qquad (23)$$

But this is simply the derivative of the magnitude response h(t). Assuming h(t) has the general form in equation (24),

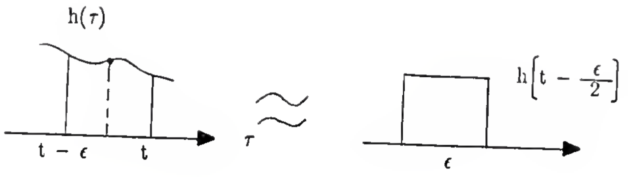
$$h(t) = K e^{-ct^2} \qquad (24)$$

the derivative of h(t) is

$$h'(t) = -2ckt\ e^{-ct^2} . \qquad (25)$$

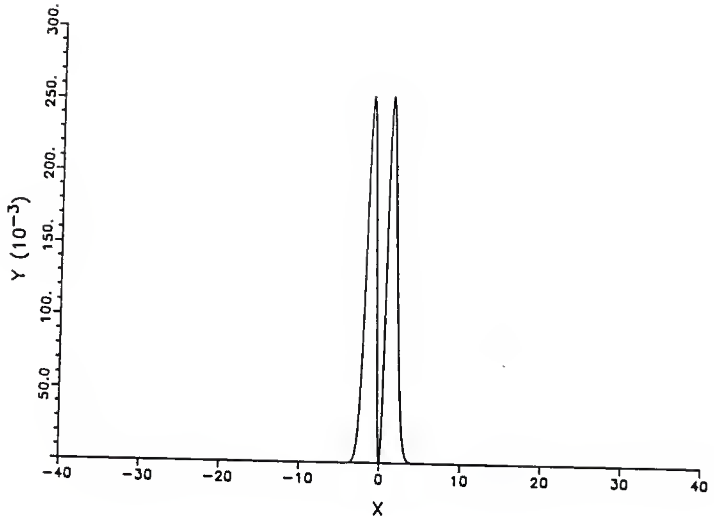
Squaring yields the magnitude response. Plot 5–11 shows the output of the frequency compressive receiver to a bi–phase long cw input as derived above. Note the resemblance to plot 5–8.

This result helps to explain some of the observed results. For phase shifts of less than 180 degrees, an "unbalanced" doublet is produced, because the positive and negative frequencies don't produce opposite impulses. This can be thought of as a combination of a weak doublet and a single impulse. The resulting output is an addition of their combined responses. Thus the notch is partially filled in. If the phase transition doesn't occur in the middle,



$$\text{area} \approx \epsilon\, h\left[t - \frac{\epsilon}{2}\right]$$

Figure 6 Approximation of the area under the integral.



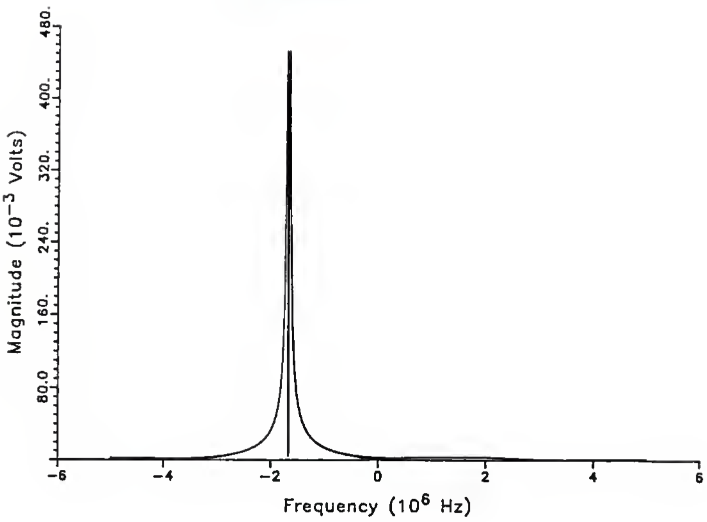
Plot 5–11 Theoretical Magnitude Response of Frequency Compressive Receiver to a BPSK input.

again an unbalanced doublet results because the opposing phases don't have equal energy. Finally, moving the input frequency to where the phase transition occurs in the input has the effect of re–centering the phase shift in the mixer output. Any receiver imperfections, such as the errors previously investigated, which prevent the formation of impulses will greatly affect the generation of a notched output.

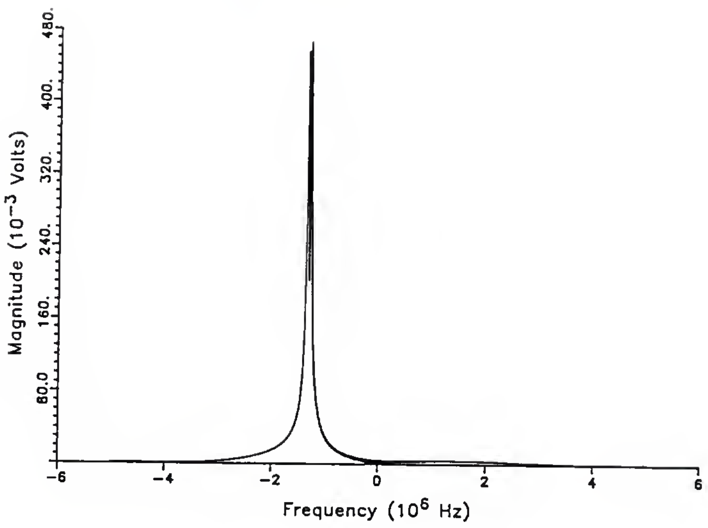
The last group of plots show the receiver output pulse for an input pulse with three phase shifts of 0, 180 and 0 degrees. What is varied is the frequency of the input pulse, or the position of the phase transition in the mixer output. Plots 5–12 through 5–19 show the receiver output for input carrier frequencies of ± 1.67, 1.3, 1.0, and 0.5 MHz. This shows how unequal divisions of phase affect the output pulse. Graph 5–3 shows probability of detection vs. frequency.

The phase notch presents a potential problem in detecting PSK pulses with a frequency compressive receiver. If the phase transitions somehow synchronize with the receiver's scanning, detection of weak PSK pulses, especially BPSK, could be difficult because of the peak reduction by the notch. Additionally, a slight frequency error occurs because the two peaks don't occur at the center of the dispersive filter response.

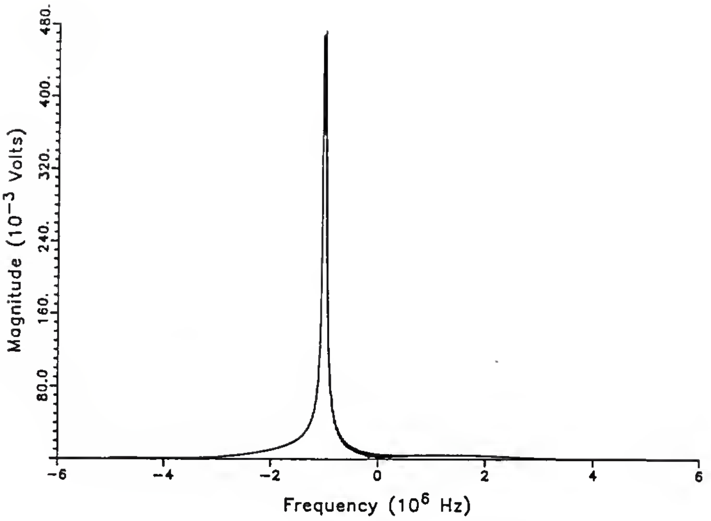
Without apriori knowledge, identifying the type of modulation, if any, used for a particular input signal could be difficult. This is because of the similarity in appearance of CW, PSK and narrowband FSK.



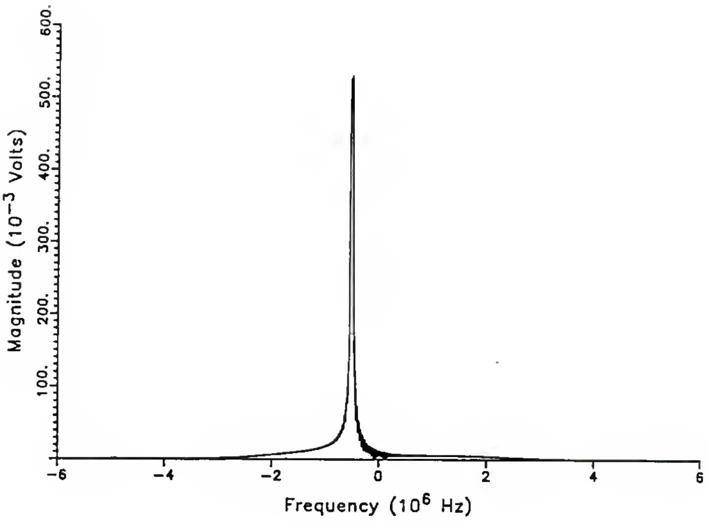
Plot 5–12 Frequency Compressive Receiver output for a three phase transition BPSK input at –1.67 MHz.



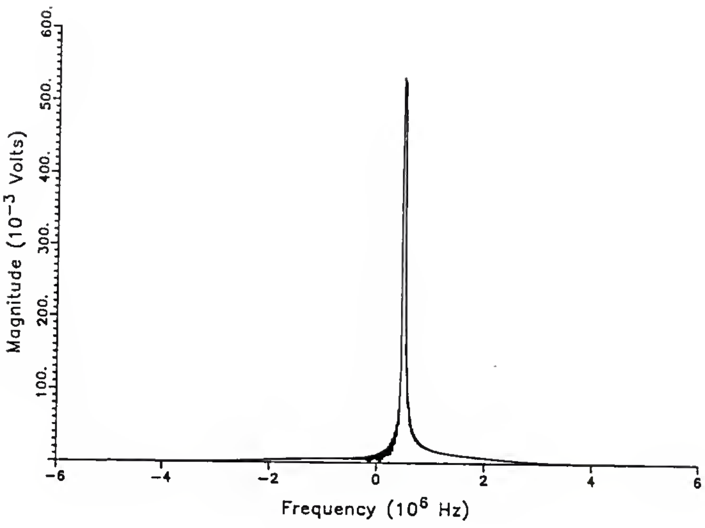
Plot 5–13 Frequency Compressive Receiver output for a three phase transition BPSK input at –1.3 MHz.



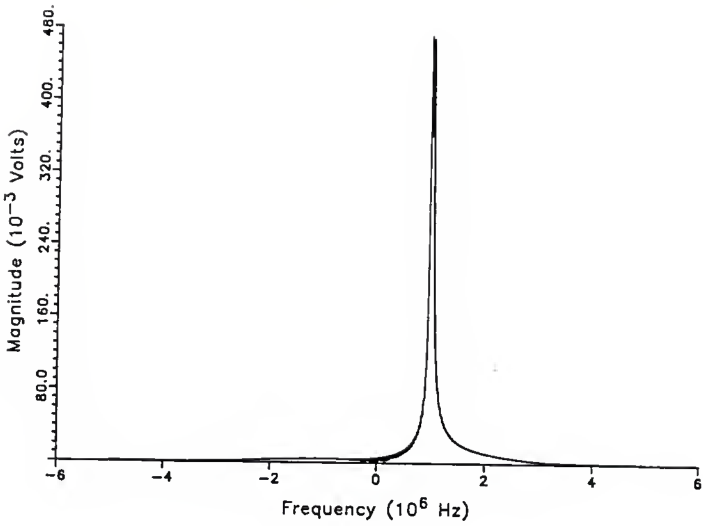
Plot 5-14 Frequency Compressive Receiver output for a three phase transition BPSK input at -1.0 MHz.



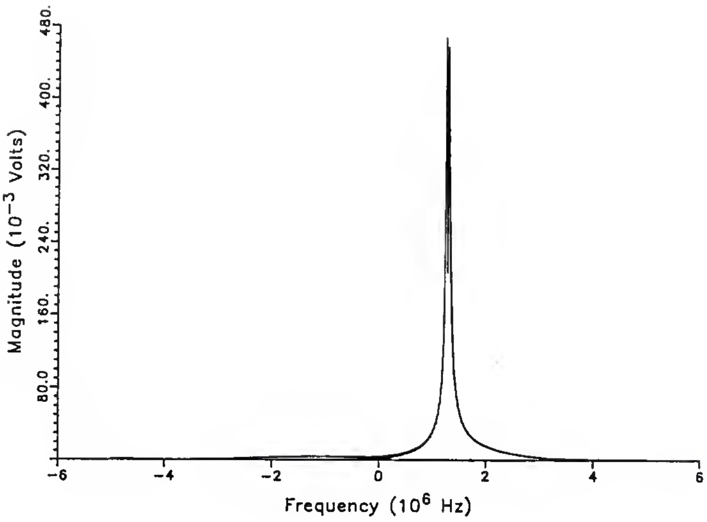
Plot 5–15 Frequency Compressive Receiver output for a three phase transition BPSK input at −0.5 MHz.



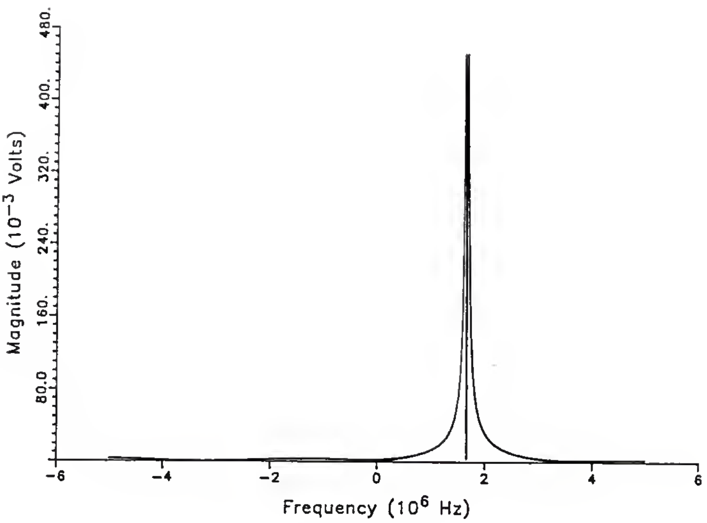
Plot 5–16 Frequency Compressive Receiver output for a three phase transition BPSK input at 0.5 MHz.



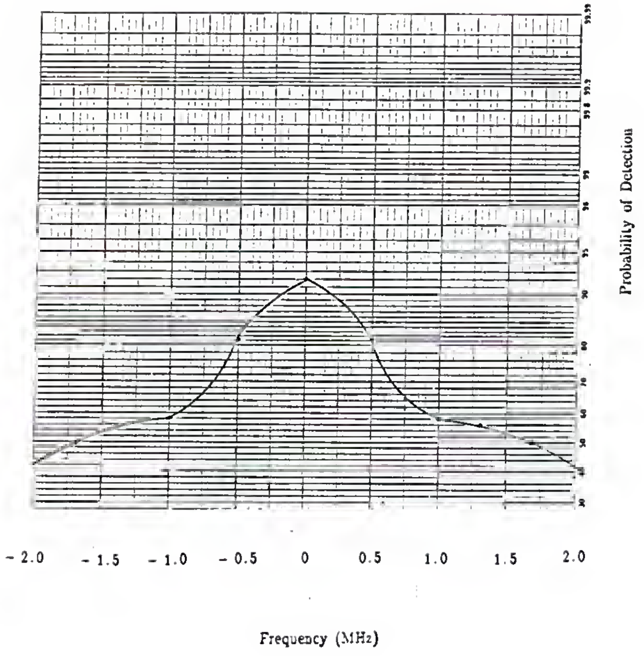
Plot 5–17 Frequency Compressive Receiver output for a three phase transition BPSK input at 1.0 MHz.



Plot 5–18 Frequency Compressive Receiver output for a three phase transition BPSK input at 1.3 MHz.



Plot 5–19 Frequency Compressive Receiver output for a three phase transition BPSK input at 1.67 MHz.



Graph 5-3 Probability of Detection vs. Frequency for three phase transition BPSK.

References

[1] B.K. Harms, "An Analysis and Comparison of the Channelized, Acoustooptic, and Frequency Compressive Intercept Receivers", Final Report to Motorola Government Electronics Group, Kansas State University, Department of Electrical and Computer Engineering, June, 1985.

[2] T.R. Walsh, "An Analysis of a Frequency Compressive Receiver", Final Report to Motorola Government Electronics Group, Kansas State University, Department of Electrical and Computer Engineering, July, 1986.

[3] B.K. Harms and D.R. Hummels, "Calculation of Detection Probability for Frequency Compressive Receivers", IEEE Trans. on Aerospace and Electronic Systems, Vol. AES–21, No. 1, January, 1985.

[4] L.E. Brennan and I.S. Reed, "A Recursive Method of Computing the Q Function", IEEE Trans. on Information Theory, Vol. IT–11, No. 2, pp. 312, April, 1965.

Appendix A

User's Guide to Simulation Software

This is a user's guide to the frequency compressive receiver simulation program. It describes overall features as well as user inputs and program outputs. An example simulation run is given at the end.

The original program was written in VAX C under the VMS operating system. This user's guide applies to the version modified for Microsoft C version 5.0 under MSDOS. In this program, many input variables and receiver parameters are hard coded into the mainline source code, rx.c. An attempt was made to throughly document all these variables, should subsequent modifications be desired.

The simulation creates two optional output files per run. The first one writes the output of the square law detector, the receiver output, to a file in the same directory as the executable file. Because the display and plotting capabilities of the user are unkown, the user can read this file and adapt his/her plotting routine accordingly. The file format is an integer on one line, followed by pairs of doubles on succesive lines. The integer is the number of complex data pairs that follow. The second output is a text file, also written to the executable's directory. It contains all the receiver parameters and simulation results and is in the same format as the screen output.

After starting the simulation, rx.exe, the first prompt asks for the number of phase values to apply to the input signal. This value, from 1 to 10, divides the input pulse into equal regions. It doesn't matter if the input pulse is long, short, rectangular, trapezoidal, or triangular. After the value has been entered, the user is then asked for the phase values, in degrees, for each region. The phases are applied in the order they are entered. If a PSK input is desired, this is where you set the phase values.

The next prompt is for the number of frequencies (1 - 10)to use to modulate the input signal. This is similar to the phase case. The frequencies for the regions are entered in Hertz. Again, the frequencies are used to modulate in the order they are entered. If an FSK input is desired, this is where the frequencies would be set.

The next group of inputs are receiver characteristic parameters. The key to the operation of the frequency compressive receiver is that the slopes of the sweeping local oscillator and dispersive filter match. The S.L.O. and the dispersive filter have the same three options for the slope used. The prompts occur for the SLO first and then the dispersive filter. The first option lets the user use the ideal, or theoretical slope for the S.L.O. and dispersive filter. It is based on receiver parameters set in the mainline. The second option allows the user to use an arbitrary polynomial with order of zero to 10. If this option is chosen, the user is asked for the order of the polynomial and then the coefficients in ascending power order. After entry, the polynomial is then integrated before use. The third and final option allows the user to enter a factor to multiply the ideal slope of the ideal equation by.

At this point, the receiver has generated a complex array output from the square law detector which represents the output of the receiver. The user is asked if the output data are to be saved, and if so, under what filename. If display or plotting of this output is desired, it needs to be saved because no plotting routine has been provided with this version of the program. Note that the program has already stored the information it needs to continue the simulation, so not saving has no effect on continued execution.

Now the program prepares to take into account the effect of noise on the input. The user is asked for the signal–to–noise density ratio in dB and for the false alarm rate in false alarms per second. Now the program begins to print the receiver parameters, all the input signal parameters, the SLO and dispersive filter parameters and finally the output statistics.

Since this is a lot of information all at once, the user may instruct the program to save this output to a data file. If the decison is yes, the user is further prompted for a file name and comments. The comments are to facilitate keeping track of different simulations results. Up to 80 characters can be entered as comments.

The final prompt asks if you would like to calculate a new probability of detection based on a new signal–to–noise ratio and false–alarm rate. This way, only part of the simulation has to be rerun for a given receiver input with changing noise levels. Answering "no" to this question terminates program execution.

For an example, let's say you wanted to have an input that was a long PSK pulse with phase values of 45 and 234 degrees (arbitrary values) at an input frequency of –1.5 MHz. Further, you want to use an ideal SLO and a dispersive filter with a quadratic time delay vs. frequency relationship. Finally, the signal to noise density ratio is 60 dB and the false alarm rate is 1 per minute. The output parameters are to be saved but not the signal itself.

The computer's screen output is enclosed in < > and the user's response immediately follows. Starting the program, you see:

```
< Frequency Compressive Rcceiver Program >

< Enter number of phase values  1 - 10 >
  2
< Enter phase for region 1 >
  45
< Enter phase for region 2 >
  234


< Enter number of frequency values  1 - 10 >
  1
< Enter frequency for region  1 >
  -1.5e6


< Select the case for Inst. frequency vs. time
      1. Ideal Case
      2. Arbitrary Polynomial
      3. Vary ideal slope by a const. multiplier >
  1
< Polynomial used:
      degree = 2
      y[0]  = 0.0
      y[1]  = -3.14e7
      y[2]  = 3.39e11  >

< Select the case for time delay vs. frequency
      1. Ideal Case
      2. Arbitrary Polynomial
      3. Vary ideal slope by a const. multiplier >
  2
< Enter the order of the polynomial >
  2
< Enter the coefficient for the 0 power >
  c
< Enter the coefficient for the 1 power >
  b
< Enter the coefficient for the 2 power >
  c
< Polynomial used:
      degree = 3
      y[0]  = 0.0
      y[1]  = c
      y[2]  = b/2
      y[3]  = c/3  >
```

Note: A, B, and C are just arbitrary values. Program integrates all arbitrary polynomials.

```
< Do you wish to save the receiver output ? y/n >
  n

< Enter the signal to noise density ratio in dB >
  60

< Enter the false alarm rate >
  1

< Then a lot of output about parameters used... >

< Do you wish to save the results ?  y/n >
  y
< Enter the filename to save under >
  trial.one
< Enter any comments for the file >
  Wow, my first ever output file!!!!

< Enter 1 to calculate a new probability  >
  9

< End program >
```

Appendix B

Computer Programs

<u>Programs in order of appearance</u>.

rx.c	Mainline code. Calls other routines and finds probability of detection. Also outputs to results to a file.
pulse.c	Generates passband input.
mixer.c	Time domain signal mixer.
filter.c	Sets up filtering operation.
gauss_flt.c	Gaussian magnitude filter.
s_law_det.c	Square law detector. Also finds peak signal and its frequency.
coverage.c	Find probability of detection using Brennen's algorithm.
fcr.h	Header file used by all routines.
poly_op.c	Polynomial operations. Evaluation and integration.
fft.c	Fast Fourier Transform.
plot_op.c	Set up titles, etc., for plots.
simple_plot.c	Plotting routine.

```
/*****************************************************************
 *
 *  SOURCE FILE:  rx.c
 *
 *
 *  DESCRIPTION:  This is the mainline code for the frequency
 *                compressive receiver project.
 *
 *
 *  FUNCTIONS
 *  CALLED:
 *                pulse
 *                mixer
 *                fft
 *                filter
 *                s_law_det
 *                prob_of_det
 *                simple_plot
 *
 *
 *  AUTHOR:       Phillip Fry
 *
 *
 *  DATE CREATED: 26JAN88           VERSION:  1.0
 *
*****************************************************************/

#include   "fcr.h"
#include   time

double      coverage();
char        *ctime();
long        *time();


main()
{
    FILE     *out_f,
             *fopen();

    COMPLEX x[MAX_SIZE],                  /* Holds complex       */
            y[MAX_SIZE];                  /* signal information  */

    POLYNOMIAL slo,
               flt,
               theta,
               freq;

    int     i,                            /* Looping variable    */
            devnum,
            tics,                         /* VAX time in seconds.*/
            n         = 1024;             /* Number of points.   */

   double   comp_bw   = 5E6,              /* Compression bw      */
            tau       = 4E-5,             /* Dispersion time.    */
```

```
        /* Compressive filter characteristics                              */
        cf_bw    = 1.8E6,                  /* RF bandwidth               */
        cf_td    = 0.0,                    /* constant time delay */
        cf_theta = 0.0 ,                   /* constant phase delay*/
        wsigma2,                           /* Variance of                */
                                           /* compressive filter  */
                                           /* output.                    */

        /* Input pulse characteristics.                                   */
        tw       =10E-6,                   /* Pulse width.               */
        tr       =0.0,                     /* Rise time of pulse  */
        td       =35E-6,                   /* Delay time of pulse */
                                           /* in L.P. model.             */


        t_step,                            /* Time inc per sample */
        scan_rate,                         /* Rate of LO scan hz/s*/
        rw,                                /* scan rate in radians*/

        sndr,                              /* S/N density ratio   */
        sndrdb,                            /* sndr in dB                 */
        sweep_time,                        /* time to cover bw    */
        fa_rate,                           /* False alarm rate.   */
        prob_fa,                           /* Probability of a    */
                                           /* false alarm.               */
        threshold,                         /* Voltage that yields */
                                           /* desired prob of FA  */
        pk_signal,                         /* Peak signal value   */
        prob_det,                          /* Prob. of detection  */

        w3,                                /* L.P. equiv 3dB bw   */
        bnrf,                              /* equiv. noise bw     */
        no,                                /* Noise density.             */

        pk_freq,                           /* Freq of filter pk   */
        s_f,                               /* Factor of SLO slope */
        f_f,                               /* Factor of flt slope.*/


        x_axis[MAX_SIZE],                  /* Data arrays for the */
        y_axis[MAX_SIZE];                  /* plotting fn.               */



char    choice,                            /* Decision variable.  */
        fn[15],                            /* Filename of output. */
        destination[30] =                  /* Directory for record*/
              {"ee:[fry.records]"},
        comments[80];                      /* File comments.      */
```

```
    printf("\7\n Frequency Compressive Receiver Program");
    printf("\nenter 4014, 7475 or 0 for no plot\n");
    scanf("%d",&devnum);

    tw = 80e-6;
    tr = 0.0;
    td = 0.0;

    sweep_time = 2 * tau;
    scan_rate  = comp_bw / tau;
    t_step     = sweep_time / n;
    rw         = 2 * PI * scan_rate;
    w3         = PI * cf_bw;



    pulse(&x,n,&freq,&theta,tw,tr,td,sweep_time,t_step);

    mixer(&x,n,scan_rate,sweep_time,t_step,rw,&slo,&s_f);

    fft(&x,&y,n,FORWARD);

    filter(&y,n,w3,rw,cf_td,cf_theta,sweep_time,&flt,&f_f);

    fft(&y,&x,n,REVERSE);

    if  ( devnum != 0 )
        plot_op(&x,n,comp_bw,devnum);

    s_law_det(&x,n,&pk_signal,comp_bw,&pk_freq);

/*-----------------------------------------------------------------*/
/* Begin performance calculations.                                 */
/*-----------------------------------------------------------------*/
    printf("\7");
    do
        {
        choice = getchar();
        printf("\n\nEnter the signal to noise density ratio:\n");
        scanf("%E",&sndrdb);
        printf("\nEnter the false alarm rate:\n");
        scanf("%E",&fa_rate);

        sndr = pow(10.0,(sndrdb / 10.0));
        no   = AMPLITUDE * AMPLITUDE / (2.0 * sndr);
        bnrf = (cf_bw / 2.0) * sqrt(PI / log(2.0));
        wsigma2 = no * bnrf;
        prob_fa = fa_rate/bnrf;
        threshold = -2.0 * wsigma2 * log(prob_fa);
        prob_det = 1.0 - coverage( sqrt(threshold),
                             sqrt(pk_signal),wsigma2,ERR );
        printf("Receiver Parameters:\n");
        printf("Number of data pts: %d\n",n);
        printf("Comp B.W.(Hz):%E\n",comp_bw);
        printf("Dispersion time:%e\n",tau);
        printf("Scan rate (Hz/s):%E\n",scan_rate);
        printf("Sweep time (secs):%e\n",sweep_time);
        printf("3 dB RF BW (Hz):%E\n",w3);

        printf("\nInput Pulse parameters:\n");
        printf("fsk values\n");
        for ( i = 0; i < freq.degree ; ++i )
            printf("freq[%d]=%e(Hz)\n",i,freq.a[i]);
        printf("\npsk values\n");
        for ( i = 0; i < theta.degree ; ++i )
            printf("theta[%d] (Degrees):%e\n",i,theta.a[i] );
        printf("\nPulse delay (s):%e\n",td);
        printf(" \"   width (s):%e\n",tw);
        printf(" \" rise time  :%e\n",tr);

        printf("\nTrial Parameters and results\n");
        printf("S/N density ratio:%e\n",sndr);
        printf("False Alarm Rate:%f \n",fa_rate);
        printf("Prob. false alarm:%e\n",prob_fa);
        printf("Threshold (Volts):%f\n",threshold);
        printf("Pk signal (volts):%f\n",pk_signal);
        printf("Freq of peak(Hz) :%E\n",pk_freq);
        printf("Prob of det :%7.6f\n",prob_det);

/*
        printf("\n Save results ?? y/n \n");
        choice = getchar();
        choice = getchar();
        scanf("%c",choice);
*/
        choice = 'y';
        if ( choice == 'y' )
            {
            printf("Enter Filename to save under\n");
            scanf("%s",fn);
            printf("Enter any comments for the file\n");
            gets(&comments);
            gets(&comments);

            strcat(destination,fn);
            out_f = fopen(destination,"w");
            if ( out_f != NULL )
                {
                time(&tics);
                fprintf(out_f,"\n%s\n",destination);
                fprintf(out_f,"%s\n", ctime(&tics) );
                for ( i = 0; i < 80 ; ++i )
                    fprintf(out_f,"%c", comments[i] );
                fprintf(out_f,"\n\n");
        fprintf(out_f,"Receiver Parameters:\n");
        fprintf(out_f,"Number of data pts: %d\n",n);
        fprintf(out_f,"Comp B.W.(Hz):%E\n",comp_bw);
        fprintf(out_f,"Dispersion time:%e\n",tau);
        fprintf(out_f,"Scan rate (Hz/s):%E\n",scan_rate);
        fprintf(out_f,"Sweep time (secs):%e\n",
                                   sweep_time);
        fprintf(out_f,"3 dB RF BW (Hz):%E\n",w3);

        fprintf(out_f,"\nInput Parameters:\n");
        fprintf(out_f,"FSK values used.\n");
for ( i = 0; i < freq.degree ; ++i )
     fprintf(out_f,"freq[%d]=%e(Hz)\n",i,freq.a[i]);
fprintf(out_f,"\npsk values\n");
for ( i = 0; i < theta.degree ; ++i )
     fprintf(out_f,"theta[%d] (Degrees):%e\n",i,theta.a[i] );
        fprintf(out_f,"\nPulse delay (s):%e\n",td);
        fprintf(out_f," \"   width (s):%e\n",tw);
        fprintf(out_f," \" rise time  :%e\n",tr);

        fprintf(out_f,"\nSLO slope factor %f\n",s_f);

        fprintf(out_f,"\nPolynomial for SLO: degree=%d\n"
                             ,slo.degree );
        for (i = 0; i <= slo.degree ; ++i )
            fprintf(out_f,"%E   ",slo.a[i] );
        fprintf(out_f,"\n\n");

        fprintf(out_f,"FILTER slope factor %f\n",f_f);

        fprintf(out_f,"\nPolynomial for filter:");
        fprintf(out_f," degree=%d\n",flt.degree );
        for (i = 0; i <= flt.degree ; ++i )
            fprintf(out_f,"%E   ",flt.a[i] );
        fprintf(out_f,"\n");


        fprintf(out_f,"\nTrial Parameters & results\n");
        fprintf(out_f,"S/N density ratio:%e\n",sndr);
        fprintf(out_f,"False Alarm Rate:%e \n",fa_rate);
        fprintf(out_f,"Prob. false alarm:%f\n",prob_fa);
        fprintf(out_f,"Threshold Volts:%f\n",threshold);
        fprintf(out_f,"Pk signal volts:%f\n",pk_signal);
        fprintf(out_f,"Freq of peak (Hz):%E\n",pk_freq);
        fprintf(out_f,"Prob of det :%7.6f\n",prob_det);

        fclose(out_f);
        }
    else
        {
        printf("\n Error in file write. ");
        printf("or disk space.\n\n");
        }
    };
```

```
        printf("\n\nEnter 1 to calculate a new probability\n");
        scanf("%c",&choice);
        scanf("%c",&choice);

        )
    while ( choice == '1' );


    printf("\n End program");

    return(0);


)
```

```
/*****************************************************************
 *
 *  SOURCE FILE:   pulse.c
 *
 *
 *  FUNCTION:      pulse(x,n,diff_freq,theta,tw,tr,td,sweep_time)
 *
 *
 *  DESCRIPTION:   This function generates an array of sampled
 *                 points from a trapezoidal pulse.
 *
 *
 *
 *  ARGUMENTS:
 *    x            (output) *COMPLEX
 *                 Complex data array with time domain pulse.
 *
 *    n            (input) int
 *                 Number of data points in x.
 *
 *    diff_freq    (output) POLYNOIMIAL
 *                 Frequency offsets for pulse in low pass model.
 *                 The degree is the number of phase values in
 *                 the coeffecient array.  In degrees.
 *
 *    theta        (output) POLYNOMIAL
 *                 The values used for the phase of the pulse.
 *                 The degree is the number of frequency values
 *                 in the coeffeceint array.  In Hertz.
 *
 *    tw           (input) double
 *                 Width of pulse.
 *
 *    tr           (input) double
 *                 Rise time of pulse.
 *
 *    td           (input) double
 *                 Pulse delay.
 *
 *    sweep_time   (input) double
 *                 Time in seconds for one sweep of twice
 *                 compressive bandwidth.
 *
 *    tstep        (input) double
 *                 Time increment per data point.
 *
 *
 *  RETURN:        Void.
 *
 *
 *  FUNCTIONS
 *  CALLED:        cmult()
 *                 cmag()
 *                 cmplx()
 *
 *
 *  AUTHOR:         Phillip Fry
 *
 *
 *  DATE CREATED:   26JAN88           VERSION:  3.0
 *
 *
 *  REVISIONS:      14JUL88  Added CW or PSK pulse option.
 *
 *                  24AUG88  Added M-Ary PSK for any type pulse.
 *
 *                  07SEP88  Added FSK option.
 *
*****************************************************************/

#include "fcr.h"


void        pulse(x,n,diff_freq,theta,tw,tr,td,sweep_time,tstep)

COMPLEX     *x;
POLYNOMIAL *diff_freq,
            *theta;
int         n;
double      tw,
            tr,
            td,
            sweep_time,
            tstep;

{
    int     i,
            j,
            start,
            stop,
            inc;

    double arg,
           mag,                /* value of lp envelop */
           temp,
           re_factor,
           im_factor;

    char   choice;


    choice = getchar();


    printf("\n Enter number of phase values 1 - 10\n");
    scanf("%d",&theta->degree);
```

```
      for ( i = 0; i < theta->degree; ++i )
          {
          printf("\nEnter phase for transition %d\n",i+1);
          scanf("%E",&theta->a[i] );
          }


      printf("\n Enter number of frequency values 1 - 10\n");
      scanf("%d",&diff_freq->degree);

      for ( i = 0; i < diff_freq->degree; ++i )
          {
          printf("\nEnter frequency for transition %d\n",i+1);
          scanf("%E",&diff_freq->a[i] );
          }




/*----------------------------------------------------------------*/
/* Initial zero.                                                  */
/*----------------------------------------------------------------*/
    i = 0;
    while ( (i * tstep) < td )
        {
        x[i].re = 0.0;
        x[i].im = 0.0;
        ++i;
        };


/*----------------------------------------------------------------*/
/* Generate rising edge.                                          */
/*----------------------------------------------------------------*/
    while ( (i * tstep) < (tr + td) )
        {
        mag = (AMPLITUDE / tr) * (i * tstep - td);
        x[i].re = mag;
        x[i].im = mag;
        ++i;
        };


/*----------------------------------------------------------------*/
/* Generate plateau.                                              */
/*----------------------------------------------------------------*/
    while ( (i * tstep) < (tw + td) )
        {
        x[i].re = AMPLITUDE;
        x[i].im = AMPLITUDE;
        ++i;
        };
```

```
/*-----------------------------------------------------------------*/
/* Generate falling edge.                                          */
/*-----------------------------------------------------------------*/
    while ( (i * tstep) < (tw + td + tr) )
        {
        mag = (tw + tr + td - i * tstep) * (AMPLITUDE / tr);
        x[i].re = mag;
        x[i].im = mag;
        ++i;
        };


/*-----------------------------------------------------------------*/
/* Generate zero tail                                              */
/*-----------------------------------------------------------------*/
    while ( (i * tstep) < sweep_time)
        {
        x[i].re = x[i].im = 0.0;
        ++i;
        };



/*-----------------------------------------------------------------*/
/* Superimpose M-Ary PSK pattern.                                  */
/*-----------------------------------------------------------------*/
    if ( choice != 'c' )
        {
        i = 0;
        while ( x[i].re == 0.0 )
            ++i;
        start = i;
        while ( x[i].re != 0.0 )
            ++i;
        stop = i - 1;

        inc = (int) ( 1 + (( stop - start )/ theta->degree)  ) ;

        for ( i = 0; i < theta->degree ; ++i )
            {
            re_factor = cos( theta->a[i] * PI/180 );
            im_factor = sin( theta->a[i] * PI/180 );
            for ( j = start ; j < start + inc ; ++j )
                {
                x[j].im = im_factor * x[j].re;
                x[j].re *= re_factor;
                }
            start += inc;
            }
        }
```

```
/*-------------------------------------------------------------*/
/* Set frequency of low pass waveform.                          */
/*-------------------------------------------------------------*/
    i = 0;
    while ( x[i].re == 0.0 )
        ++i;
    start = i;

    inc = (int)(1 + ((stop - start)/diff_freq->degree));

    for (i = 0; i < diff_freq->degree; ++i)
        {
        temp = 2 * PI * diff_freq->a[i] * tstep;
        for ( j = start; j < start + inc; ++j )
            {
            arg = temp * j;
            x[j] = cmult( x[j], cmplx( cos(arg),sin(arg) ) );
            }
         start += inc;
         }


    return;
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:   mixer.c
 *
 *
 *  FUNCTION:      mixer(x,n,scan_rate,sweep_time,tstep,rw,vst,
 *                                factor)
 *
 *
 *  DESCRIPTION:   Mixes input pulse with SLO.  There are 5
 *                 options for SLO source.
 *                  1. Ideal
 *                  2. Polynomial model
 *                  3. Data point model using polynomial
 *                     interpolation.
 *                  4. Linear slope error.
 *                  5. S.L.O. phase noise.
 *
 *
 *  ARGUMENTS:
 *     x           (input/output) *COMPLEX
 *                 Signal in, I.F. out.
 *
 *     n           (input) int
 *                 Number of points in x.
 *
 *     scan_rate   (input) double
 *                 Rate of LO scan, Hz/sec.
 *
 *     sweep_time  (input) double
 *                 Time to cover compressive bandwidth.
 *
 *     tstep       (input) double
 *                 Time increment per sample.
 *
 *     rw          (input) double
 *                 Scan rate in radians.
 *
 *     vst         (output) *POLYNOMIAL
 *                 Polynomial used to calculate the phase.
 *
 *     factor      (output) *double
 *                 A value that multiplies the slope of t vs f.
 *
 *
 *  RETURN:        Void.
 *
 *
 *  FUNCTIONS
 *  CALLED:        cmult()
 *                 cmplx()
 *
 *
 *  AUTHOR:        Phillip Fry
 *
 *
 *  DATE CREATED:  30JAN88           VERSION:  1.0
 *
 *
 *  REVISIONS:     15JUN88    Added options on SLO source.
 *                 22JUN88    Added slope factor option.
 *                 30SEP88    Added phase noise option.
 *
****************************************************************/

#include "fcr.h"

double   eval_poly();

void     mixer(x,n,scan_rate,sweep_time,tstep,rw,vst,factor)

POLYNOMIAL *vst;
COMPLEX    *x;
int        n;
double     scan_rate,
           sweep_time,
           tstep,
           *factor,
           rw;
{
    int    i,                  /* Looping variable.            */
           seed;

    double phi,                /* Quadratic phase term.        */
           mean,
           var;

    COMPLEX noise[1024];

    char   choice;             /* Decision variable.           */


    choice = getchar();
    choice = getchar();
    printf("\nSelect the case for Inst. freq vs time \n");
    printf("  1. Ideal \n  2. Arbitrary Polynomial");
    printf("\n  3. Polynomial from interpolated data points\n");
    printf("  4. Vary ideal slope by a constant multiplier\n");
    printf("  5. Gaussian phase noise added to ideal S.L.O.\n");
    scanf("%c",&choice);
    scanf("%c",&choice);

    switch ( choice )
        {
        case '3':
            /* call subroutine that returns a polynomial */
            integ_poly(vst);
            break;
        case '2':
            printf("Enter the order of the polynomial\n");
            scanf("%d",&vst->degree);
            printf("\n");
            for ( i = 0; i <= vst->degree ; ++i )
                {
                printf("Enter coeffecient for %dth power\n",i);
                scanf("%E",&vst->a[i]);
                }
            integ_poly(vst);
            break;
        case '4':
            printf("\nEnter the slope factor. \n");
            scanf("%E",factor);
        case '1':
        case '5':
            if ( choice != '4' )
                *factor = 1.0;
            vst->degree = 2;     /* already in integrated form */
            vst->a[0] = 0.0;
            vst->a[1] = -scan_rate * sweep_time * PI;
            vst->a[2] = .5 * rw * (*factor);
            break;
        default:
            printf("invalid input\n");
            return;
            break;
        }

    printf("\npolynomial used\n degree = %d\n",vst->degree);
    for ( i = 0; i <= vst->degree; ++i )
        printf("y[%d] = %E\n",i,vst->a[i] );


    if ( choice == '5' )
        {
        printf("\n\nEnter mean and variance \n");
        scanf("%E%E",&mean,&var);
        printf("Enter seed for random number generator\n");
        scanf("%d",seed);
        phase_noise(&noise,n,var,mean,seed);
        }
    else
        {
        for ( i = 0; i < n; ++i )
            noise[i] = cmplx( 0.0, 0.0 );
        }
    for (i = 0; i < n; ++i)
        {
        phi = eval_poly(vst, i * tstep ) ;
        x[i] = cmult( x[i],cmplx( cos(phi + noise[i].re ),
            -sin(phi + noise[i].im) ));
        };
    return;
}
```

```
/******************************************************************
 *
 *  SOURCE FILE:    filter.c
 *
 *
 *  FUNCTION:       filter(x,n,w3,rw,cf_td,cf_theta,factor)
 *
 *
 *  DESCRIPTION:    This function calls a Gaussian filter.  There
 *                  are 3 choices for a time delay vs t function.
 *                   1. Ideal
 *                   2. Arbitrary polynomial.
 *                   3. Polynomial from interpolated data points.
 *
 *
 *  ARGUMENTS:
 *    x             (input/output) *COMPLEX
 *                  Complex data array holding the signal.
 *
 *    n             (input) int
 *                  Number of data points in x.
 *
 *    w3            (input) double
 *                  L.P. 3dB equivalent bandwidth.
 *
 *    rw            (input) double
 *                  Scan frequency in radians.
 *
 *    cf_td         (input) double
 *                  Compressive filter time delay.
 *
 *    cf_theta      (input) double
 *                  Compressive filter phase shift.
 *
 *    y             (output) *POLYNOMIAL
 *                  Polynomial used for time delay vs freq.
 *
 *    factor        (output) *double
 *                  Used to vary the slope of w vs td by a
 *                  constant factor.
 *
 *
 *  RETURN:         Void.
 *
 *
 *  FUNCTIONS
 *  CALLED:
 *                  gauss_flt()
 *                  cmult()
 *                  integ_poly()
 *
 *
 *  AUTHOR:         Phillip Fry
 *
 *
 *  DATE CREATED: 03FEB88            VERSION:  1.0
 *
 *
 *  REVISIONS:     15JUN88   Added polynomial opts.
 *                 21JUN88   Revised t.d. calc. loop for
 *                           assymetrical fns.
 *                 22JUN88   Added slope variation option.
 *
*****************************************************************/

#include "fcr.h"

COMPLEX   gauss_flt();

double    eval_poly();


void      filter(x,n,w3,rw,cf_td,cf_theta,sweep_time,y,factor)

POLYNOMIAL *y;
COMPLEX    *x;
int        n;
double     w3,
           rw,
           cf_td,
           cf_theta,
           sweep_time,
           *factor;
{
   int    i,             /* Loop variable.                    */
          devnum;

   double w,             /* Frequency of delay time.          */
          xax[1024],
          yax[1024];

   char   choice;


   printf("\nSelect the case for time delay vs frequency. \n");
   printf("\n 1. Ideal Case \n 2. Arbitrary Polynomial");
   printf("\n 3. Polynomial from interpolated data points.\n");
   printf(" 4. Vary ideal slope by a constant factor. \n");
   scanf("%*c%c",&choice);

   switch (choice)
      {
      case '3':
          /* call subroutine that returns a poly */
          integ_poly(y);
          break;
      case '2':
          printf("\nEnter the order of the polynomial\n");
          scanf("%d",&y->degree);
          printf("\n");
```

```
            for (i = 0; i <= y->degree ; ++i )
                {
                printf("Enter coeffecient for  %dth power\n",i);
                scanf("%E",&y->a[i]);
                }
            integ_poly(y);
            break;
        case '4':
            printf("\nEnter slope factor\n");
            scanf("%E",factor);
        case '1':
            if ( choice != '4' )
                *factor = 1.0;
            y->degree = 2;
            y->a[0] =  cf_theta;
            y->a[1] = cf_td;
            y->a[2] = *factor * 1.0 / ( -2.0 * rw );
            break;
        default:
            printf("\nInvalid choice \n");
            return;
            break;
        }

    printf("\ndegree = %d\n",y->degree );
    for ( i = 0; i <= y->degree; ++i)
        printf("y[%d] = %E\n",i,y->a[i] );




        /* Positive frequencies evaluated.       */
    for (i = 0; i <= (n/2)  ; ++i)
        {
        w = i * ( 2.0 * PI / sweep_time );
        x[i] = cmult( x[i], gauss_flt( w,w3,y) );
        }

        /* Negative frequencies evaluated.       */
    for ( i = n-1 ; i >= (int)(1 + (n/2)) ; i-- )
        {
        w = -( n - i ) * ( 2.0 * PI ) / sweep_time;
        x[i] = cmult( x[i], gauss_flt( w,w3,y) );
        };


    return;
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:     gauss_flt.c
 *
 *
 *  FUNCTION:        gauss_flt(w,w3,x)
 *
 *
 *  DESCRIPTION:     This function models an ideal Gaussian filter.
 *
 *
 *  ARGUMENTS:
 *      w            (input) double
 *                   Frequency, in rads, at which to evaluate
 *                   at.
 *
 *      w3           (input) double
 *                   L.P. 3dB bandwidth equivalent frequency.
 *
 *      x            (output) *POLYNOMIAL
 *                   Polynomial used for time delay.
 *
 *
 *  RETURN:          Complex
 *                   Response of filter at w.
 *
 *
 *  FUNCTIONS
 *  CALLED:          cmplx()
 *                   eval_poly()
 *
 *
 *  AUTHOR:          Phillip Fry
 *
 *
 *  DATE CREATED:    02FEB88        VERSION 1.0
 *
 *
 *  REVISIONS:       15JUN88      Passing a poly for time delay.
 *
*****************************************************************/

#include "fcr.h"

double      eval_poly();

COMPLEX     gauss_flt(w,w3,x)
POLYNOMIAL *x;

double      w,
            w3;

{
    double  arg,            /* Quadratic phase response term. */
            scale;          /* Magnitude response.            */
    COMPLEX temp;

    int     i;


    arg = eval_poly(x,w);
    scale = exp( -.346574 * ((w * w)/(w3 * w3)));

    temp = cmplx( scale * cos(arg),scale * sin(arg) );

    return(temp);
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:    s_law_det.c
 *
 *
 *  FUNCTION:       s_law_det(x,n,pk_signal,comp_bw,pk_freq)
 *
 *
 *  DESCRIPTION:    Square law detector of filter ouput. Finds
 *                  the peak output value and the frequency
 *                  where it occurs.
 *
 *
 *  ARGUMENTS:
 *     x            (input/output) *COMPLEX
 *                  Data to be magnitude squared.
 *
 *     n            (input) int
 *                  Number of data points in x.
 *
 *     pk_signal    (output) *double
 *                  The square root of the largest value
 *                  that x attains.
 *
 *     comp_bw      (input)  double
 *                  Bandwidth of frequency compression.
 *
 *     pk_freq      (output) *double
 *                  Frequency where pk_signal occurs.
 *
 *  RETURN:         Void.
 *
 *
 *  MACRO
 *  CALLED:         MAX()
 *
 *
 *  AUTHOR:         Phillip Fry
 *
 *
 *  DATE CREATED: 30JAN88             VERSION:  1.0
 *
 *
 *  REVISIONS:     15JUN88   Added pk_freq operation.
 *
 *
*****************************************************************/

#include  "fcr.h"


void     s_law_det(x,n,pk_signal,comp_bw,pk_freq)

COMPLEX  *x;
int      n;
```

```
double    *pk_signal,
          *pk_freq,
          comp_bw;
{
    int i;


    for (i = 0; i < n ; ++i)
        {
        x[i].re = pow( cmag(x[i]) , 2.0 );
        *pk_signal = MAX(*pk_signal,x[i].re);
        x[i].im = 0.0;
        };

    i = 0;
    while ( (i < n) AND (*pk_signal != x[i].re) )
        ++i;

    *pk_freq = 2.0 * (( -n/2.0) + i) * comp_bw / (double)n ;

    return;
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:     coverage.c
 *
 *
 *  FUNCTION:        coverage(rad,disp,wsimga2,err)
 *
 *
 *  DESCRIPTION:     This function caluculates the coverage
 *                   function as defined in Patel and Read,
 *                   Handbook of the Norma Distribution.  The
 *                   results are used to find probability of
 *                   detection.
 *
 *
 *  ARGUMENTS:
 *     rad           (input) double
 *                   Radius of cover circle.
 *
 *     disp          (input) double
 *                   Displacement of circle center from origin.
 *
 *     sigma2        (input) double
 *                   Variance of distribution. (bivariate)
 *
 *     err           (input) double
 *                   Stopping criteria.
 *
 *  RETURN:          double
 *                   P( input < Threshold )
 *
 *
 *  AUTHOR:          Phillip Fry
 *
 *
 *  DATE CREATED:    04FEB88            VERSION:  1.0
 *
 *
 *  REVISIONS:
 *
 *
*****************************************************************/

#include "fcr.h"


double        coverage(rad,disp,sigma2,err)

double        rad,
              disp,
              sigma2,
              err;
{
    double
```

```
            g,                        /* Recursive summation terms*/
            k,
            sum,                      /* Resulting probability.   */
            r2,                       /* New radius.              */
            d2,                       /* New displacement.        */
            er2,                      /* Prob fa ??????           */
            temp;                     /* Intermediate summation   */
                                      /* term.                    */

    int     i,                        /* Looping variables        */
            n,
            n1,                       /* Value on n to meet error */
            nstop;                    /* Overflow value of n.     */


    r2  = pow(rad, 2.0)/(2.0 * sigma2);
    er2 = exp(-r2);
    d2  = pow(disp,2.0)/(2*sigma2);
    nstop = (int)(38.0 / log10(r2));
    n1  = (int)((rad * disp) / (sqrt(2.0) * sigma2 )) + 1;

/*
printf("\n\nn1 = %d:                      nstop = %d",n1,nstop);
printf("\nr2 = %f:   er2 = %f:   d2 = %f\n",r2,er2,d2);
*/

nstop = 400;

    if ( n1 > nstop )
        {
        printf("\nRequired iterations exceeds %d",nstop);
        printf("\nError = %f", g*k );
        return(sum);
        };

    g = 1.0 - er2;
    k = exp(-d2);
    sum = g*k;


    if ( k < 1.0e-38 ) /* VAX Fortran restriction ? */
        {
        printf("\nUnderflow error on K - disp too large");
        return( 0.0 );
        };


/* Series error aproximation invalid until n > n1      */
    for ( n=1; n <= n1 ; ++n)
        {
        temp = 1.0;
        for ( i=1; i <= n; ++i)
            temp *= r2/i;
        g -= er2 * temp;
        k *= d2/n;
        sum += g*k;
        );


    for ( n= n+1; n <= nstop; ++n)
        (
        temp = 1.0;
        for ( i=1; i <= n; ++i)
            temp *= r2/i;
        g -= er2 * temp;
        k *= d2/n;
        sum += g*k;
        if ( g*k < err )
            (
/*            printf("\nNormal termination: n=%d",n);
            printf("\n sum = %7.6f",sum);                        */
            return(sum);
            );
        );

    printf("\nRequired iterations exceeds %d",nstop);
    printf("\nError = %f", g*k );
    return(sum);

)
```

```
/*****************************************************************
 *
 *  SOURCE FILE:     fcr.h
 *
 *  DESCRIPTION:     This is the header file used with the
 *                   frequency compressive receiver project for
 *                   Motorola Goverment Electronics.
 *
 *
 *  AUTHOR:          Phillip Fry
 *
 *
 *  DATE CREATED:    26JAN88          VERSION:  1.0
 *
 *
 *  REVISIONS:
 *
*****************************************************************/

#include  <stdio.h>
#include  <math.h>
#include  "complex.h"
#include  "p_plot.h"

#define   PI           3.141592654
#define   MAX_SIZE     9000
#define   MAX_DEG      20
#define   FORWARD      1
#define   REVERSE      -1
#define   AMPLITUDE    .1
#define   ERR          1E-8

#define   AND          &&


#define   MIN( a,b )   (((a) < (b)) ? (a) : (b))
#define   MAX( a,b )   (((a) > (b)) ? (a) : (b))


typedef   struct polynomial
               {
               int        degree;

               double     a[MAX_DEG];
               }   POLYNOMIAL;
```

```
/***************************************************************
 *
 *  SOURCE FILE:    poly_op.c
 *
 *
 *  FUNCTIONS:      integ_poly(x)
 *                  eval_poly(x,k)
 *
 *
 *  DESCRIPTION:    The first function integrates the polynomial
 *                  x.  The second function evaluates the
 *                  polynomial x for the value k.
 *
 *
 *  ARGUMENTS:
 *       x          (input/output) *POLYNOMIAL
 *                  The polynomial to be operated on.
 *
 *       k          (input) double
 *                  The value to be used to evaluate x.
 *
 *
 *  RETURN:         Void:        integ_poly().
 *                  double:      Result of eval_poly().
 *
 *
 *  AUTHOR:         Phillip Fry
 *
 *
 *  DATE CREATED:   14JUN88           VERSION:  1.0
 *
***************************************************************/

#include "fcr.h"


void    integ_poly(x)
POLYNOMIAL  *x;
{
    int    i;

    for ( i = x->degree; i >= 0; i-- )
        x->a[ i + 1 ] = x->a[i]/ (double)( i + 1 );

    x->a[0] = 0.0;
    x->degree++;

    return;
}



double    eval_poly(x,k)
POLYNOMIAL *x;
```

```
double      k;
{
    int     i;

    double sum;

    for ( i = x->degree - 1, sum = x->a[ x->degree]; i>=0; i-- )
        sum = x->a[i] + sum * k;

    return(sum);
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:    fft.c
 *
 *
 *  FUNCTION:       fft(x,y,length,sign)
 *
 *
 *  DESCRIPTION:    Performs the Fast-Fourier Transform (FFT) on a
 *                  sequence of data. Also, the Inverse Fast-
 *                  Fourier Transform (IFFT) can be performed.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None.
 *
 *
 *  ARGUMENTS:
 *
 *      x           (input) *COMPLEX
 *                  Input data sequence to be manupulated.
 *
 *
 *      y           (output) *COMPLEX
 *                  Resulting output sequence after a FFT or an
 *                  IFFT has been performed on the data x.
 *
 *
 *    length        (input) int
 *                  Length of complex data arrays x and y.
 *
 *
 *    sign          (input) int
 *                  Desired transformation choice.
 *                  1 => forward fft  -1 => inverse fft
 *
 *
 *  RETURN:         int
 *                  NORMAL  : no errors were detected during the
 *                            transformation.
 *                  ERR_FFT : an error has occurred during the
 *                            transformation.
 *
 *
 *  FUNCTIONS
 *  CALLED:         None.
 *
 *
 *  AUTHOR:         Khiem D. Dao and Phillip Fry
 *
 *
 *  DATE CREATED:   14Feb88           VERSION:  2.0
 *
 *
```

```
*  REVISIONS:      26FEB88    Revised to handle complex array
*                             input and output.
*
*****************************************************************/


#include  "fcr.h"


int        fft (x,y,length,sign)
COMPLEX    *x,*y;
int        length,sign;

{
    int      i,
             j,
             k,
             m,
             n,
             mmax,
             q,
             istep;

    double   wr,
             wi,
             wpr,
             wpi,
             wtemp,
             theta,
             data[2*MAX_SIZE],
             tempr,
             tempi;



    n = 2 * length;
    j = 1;


/*------------------------------------------------------------------*/
/* Changing input data format to original complex number format.   */
/*------------------------------------------------------------------*/
    for (k = 0, q = 0; k < n; k += 2, ++q)
        {
        data[k] = x[q].re;
        data[k+1] = x[q].im;
        }


/*------------------------------------------------------------------*/
/* Begin FFT                                                        */
/*------------------------------------------------------------------*/
    for (i = 1; i <= n; i +=2)
        {
        if (j > i)
```

```
                {
                tempr = data[j-1];
                tempi = data[j];
                data[j-1] = data[i-1];
                data[j] = data[i];
                data[i-1] = tempr;
                data[i] = tempi;
                }
            m = n / 2;

            while ((m >= 2) AND (j > m))
                {
                j = j - m;
                m = m / 2;
                };
            j = j + m;
            }

        mmax = 2;
        while (n > mmax)
            {
            istep = 2 * mmax;
            theta = (2.0 * PI) / ((double)(sign * mmax));
            wpi = sin(theta);
            wpr = - 2.0 * sin(theta / 2.0) * sin(theta / 2.0);
            wr = 1.0;
            wi = 0.0;

            for(m = 1; m <= mmax; m+=2)
                {
                for(i = m; i <= n; i += istep)
                    {
                    j = i + mmax;
                    tempr = (wr * data[j-1]) - (wi * data[j]);
                    tempi = (wr * data[j]) + (wi * data[j-1]);
                    data[j-1] = data[i-1] - tempr;
                    data[j] = data[i] - tempi;
                    data[i-1] = data[i-1] + tempr;
                    data[i] = data[i] + tempi;
                    }

                wtemp = wr;
                wr = (wr * wpr) - (wi * wpi) + wr;
                wi = (wi * wpr) + (wtemp * wpi) + wi;
                }
            mmax = istep;
            }


/*-----------------------------------------------------------------*/
/* Performs scaling by 1/length if necessary and converts format.*/
/*-----------------------------------------------------------------*/
      if ( sign == FORWARD )
          for(k = 0, q = 0; k < n  ; k += 2, ++q)
              y[q] = cmplx(data[k], data[k+1]);
```

```
    else
        for(k = 0, q = 0; k < n  ; k += 2, ++q)
            y[q] = cdiv(cmplx(data[k], data[k+1]),
              cmplx(((double)length),0.0));


    return;
}
```

```
/*******************************************************************
 *
 *  SOURCE FILE:    plot_op.c
 *
 *
 *  FUNCTION:       plot_op(x,n,comp_bw,dev)
 *
 *
 *  DESCRIPTION:    This function handles the control of simple
 *                  plot and other plotting bookkeeping.
 *
 *
 *  ARGUMENTS:
 *      x           (input) *COMPLEX
 *                  Input data from the filter output,time
 *                  domain.
 *
 *      n           (input) int
 *                  Number of points in x.
 *
 *      comp_bw     (input) double
 *                  Frequency range of rx.
 *
 *      dev         (input) int
 *                  Device number to specify destination.
 *                    4014 for CRT    7475 for HP plotter.
 *
 *  RETURN:         Void.
 *
 *
 *  FUNCTIONS
 *  CALLED:         simple_plot()
 *                  cmag()
 *
 *
 *  AUTHOR:         Phillip Fry
 *
 *
 *  DATE CREATED:   15JUN88         VERSION:  1.0
 *
*******************************************************************/

#include "fcr.h"

double   cmag();

void     plot_op(x,n,comp_bw,dev)

COMPLEX  *x;
double   comp_bw;
int      n,
         dev;

{
    int  i;
    char    title[30];

    double  xaxis[MAX_SIZE],
            yaxis[MAX_SIZE];

    printf("\nEnter the plot title\n");
    gets(&title);
    gets(&title);


    for ( i = 0; i < n; ++i )
        {
        xaxis[i] = 2.0 * (( -n/2.0) + i ) * comp_bw / (double) n;
        yaxis[i] = cmag( x[i] );
        }

    simple_plot(n,xaxis,yaxis,"Frequency","Hz","Magnitude",
          "Volts",title,1,dev);

    return;
}
```

```
/*****************************************************************
 *
 *  SOURCE FILE:    SIMPLE_PLOT.C
 *
 *
 *  FUNCTION:       SIMPLE_PLOT(num_pts,x_data,y_data,x_axis_title,
 *                          x_axis_units,y_axis_title,
 *                          y_axis_units,plot_title,plot_type,
 *                          device)
 *
 *
 *  DESCRIPTION:    THIS FUNCTION RECEIVES X Y DATA AND GRAPHS THE
 *                  DATA ON EITHER THE VT100 OR TO AN HP PLOTTER.
 *                  THE FUNCTION AUTOMATICALLY SCALES THE DATA AS
 *                  TO FILL THE GRAPH.  FOUR KINDS OF PLOT FORMATS
 *                  ARE AVAILABLE: LINEAR, LOG_LINEAR, LINEAR_LOG
 *                  AND LOG_LOG.
 *
 *
 *
 *  DOCUMENTATION
 *  FILES:          NONE.
 *
 *
 *  ARGUMENTS:
 *
 *   num_pts        (input)  int
 *                  Number of X and Y data points
 *
 *   x_data         (input) double
 *                  Pointer to an array of X axis data
 *
 *   y_data         (input) double
 *                  Pointer to an array of Y axis data
 *
 *   x_axis_title   (input) char
 *                  An array that holds the X axis title label
 *
 *   x_axis_units   (input) char
 *                  An array that holds the X axis units label
 *
 *   y_axis_title   (input) char
 *                  An array that holds the Y axis title label
 *
 *   y_axis_units   (input) char
 *                  An array that holds the Y axis units label
 *
 *   plot_title     (input) char
 *                  An array that holds the title for the plot
 *
 *   plot_type      (input) int
 *                  A number that specifies the plot type
 *
 *                       LINEAR      1
 *                       LOG_LINEAR  2
*                        LINEAR_LOG   3
*                        LOG_LOG      4
*
*   devnum          (input) int
*                   Specifies the plotting device used.
*
*                   PLOTTING DEVICE:      DEVICE NUMBER ( = DEVNUM)
*                       HP 7475 PLOTTER                  7475
*                       HP 7470 PLOTTER                  7470
*                       TEKIRONIX 4014 DISPLAY           4014
*
*
*  RETURN:          int
*                   SIMPLE_PLOT() returns a flag, indicating a
*                   succesful functon execution or that a problem
*                   was encountered during execution.  Problems
*                   that set the flag are: plotter initialization
*                   error, negative numbers in a logarithmic plot
*                   or undefined plot type.
*
*
*  FUNCTIONS
*  CALLED:
*
* <<< NOTE: additional info on the following functions can be >>
* <<<     found in the P_PLOT documentation                      >>
*
*                  p_init
*                  p_clrsc
*                  p_clansi
*                  p_stvel
*                  p_orig
*                  p_linsc
*                  p_axis
*                  p_linlin
*                  p_logsc
*                  p_lgaxis
*                  p_loglin
*                  p_linlog
*                  p_loglog
*                  p_pltcnr
*                  p_stchr
*                  p_textc
*                  p_closp
*
*
*  AUTHOR:         PHILLIP FRY
*
*  DATE CREATED:   20SEP87       :VERSION 1.00
*
*  REVSIONS:
*
*   24OCT87        Added plotting device to argument list.
*
*   12NOV87        Incresed DES_TICSP to 2.50 cm
```

```
 *                  Suppresed first label on Y axis.
 *
 *   16NOV87        Added in function error messages.
 *
 *   09MAR88        Updated to handle new p_plot formats.
 *
******************************************************************/


/* REQUIRED EXTERNAL FILES FOR FUNCTION OPERATION                    */

#include    <stdio.h>
#include    "p_plot.h"


/* CONTROL CODES USED IN THE FUNCTION AND THE CALLING ROUTING    */

#define     ERROR       1         /* VALUE RETURNED ON ERROR CONDX*/
#define     NORMAL      0         /* VALUE RETURNED ON NORMAL CNDX*/
#define     LINEAR      1         /* PLOT TYPE CONTROL CODES       */
#define     LOG_LINEAR  2
#define     LINEAR_LOG  3
#define     LOG_LOG     4


/* ININITIALIZATION OF PLOTTING DEVICE PARAMETERS                    */

#define     FACTOR      1.0     /* SCALING FACTOR OF USER UNITS  */
#define     SIZE        'A'     /* SIZE OF PLOTTING PAPER        */
#define     PFILE       ".DEF"  /* SENDS PLOT DIRECTLY TO DEVICE */
#define     SYMBOL      '.'     /* PLOTTING SYMBOL (IF DESIRED)  */
#define     VELOCITY    5.0     /* VELOCITY OF PLOTTING PEN CM/S */


/*  PARAMETERS FOR THE GRAPH'S PHYSICAL LAYOUT                       */

#define     P_X_ORIG    4.1      /* PHYSICAL X ORIGIN LOCATION   */
#define     P_Y_ORIG    4.1      /* PHYSICAL Y ORIGIN LOCATION   */
#define     X_ORIG      0.0      /* USER UNITS X ORIGIN LOCATION */
#define     Y_ORIG      0.0      /* USER UNITS Y ORIGIN LOCATION */
#define     X_AXIS_LEN  18.0     /* X AXIS LENGTH IN CM          */
#define     Y_AXIS_LEN  12.0     /* Y AXIS LENGTH IN CM          */
#define     X_ANGLE     00.0     /* ANGLE OF X AXIS IN DEGREES   */
#define     Y_ANGLE     90.0     /* ANGLE OF Y AXIS IN DEGREES   */


#define     SPACING     1        /* NO. PTS SKIPPED BETWEEN DATA */
#define     DES_TICSP   2.0      /* DESIRED SPACING OF TICS IN CM*/
#define     SYM_CNTL    0        /* COMMAND FOR A LINE GRAPH     */
#define     CHR_WIDTH   .6       /* WIDTH OF LETTERS IN TITLE(CM)*/
#define     CHR_HEIGHT  .8       /* HEIGHT OF LETTERS IN CM      */
#define     CHR_ANGLE   00.0     /* ANGLE OF LETTERS IN DEGREES  */
```

```
#define    TITLE_X      9.0        /* TITLE X COORD. IN USER UNITS */
#define    TITLE_Y      12.5       /* TITLE Y COORD. IN USER UNITS */

#define    AND          &&




int   simple_plot (num_pts,x_data,y_data,x_axis_title,
                   x_axis_units,y_axis_title,y_axis_units,
                   plot_title,plot_type,devnum)

int      num_pts,plot_type,devnum;

double  x_data[],y_data[];

char     x_axis_units[],x_axis_title[],y_axis_units[],
         y_axis_title[],plot_title[];

{

    double first_x,                 /* VALUE USED TO LABEL THE 1ST */
           first_y,                 /* POINT ON AN AXIS.           */
           delta_x,                 /* VALUE USED AS AN INCREMENT  */
           delta_y,                 /* BETWEEN ADJACENT TICS.      */
           x_tic_space,             /* VALUE IN USER UNITS OF THE  */
           y_tic_space;             /* SPACING BETWEEN TIC.        */




/*------------------------------------------------------------*/
/*                                                            */
/*  THIS BLOCK INITIALIZES THE PLOTTING DEVICE AND CHECKS FOR */
/*  INIALIZATION ERRORS.  IT THEN CLEARS THE TERMINAL'S SCREEN */
/*  OR SETS THE PEN PLOTTER VELOCITY, DEPENDING ON THE PLOTTING */
/*  DEVICE CHOSEN.  FINALLY, THE PHYSICAL X AND Y ORIGINS ARE */
/*  SET FOR THE SUBSEQUENT PLOTTING OPERATIONS.               */
/*                                                            */
/*                                                            */
/*------------------------------------------------------------*/
/*
    if ( (devnum != 4014) AND (devnum != 7475) AND (devnum != 7470))
       {
       printf("\n Incorrect device type selected");
       return(ERROR);
       };
*/
    if (NORMAL != p_init(devnum ,PFILE ,FACTOR ,SIZE))
        {
        printf("\n Device initialization error: check plotter");
        return(ERROR);
        };
    if (devnum == 4014 )
        {
        p_clrsc();
        p_clansi();
        }
    else
        p_stvel ( VELOCITY );

    p_orig ( P_X_ORIG , P_Y_ORIG);


/*----------------------------------------------------------------*/
/*                                                                */
/*  THIS BLOCK DIRECTS THE PLOT DATA TO THE CORRECT PLOT TYPE    */
/*  OPERATION.  EACH OPERATION PERFORMS SCALING ON THE X AND Y   */
/*  DATA, PRODUCES THE SCALED AND LABELED AXES AND THEN CALLS    */
/*  THE APPROPIATE PLOTTING FUNCTION.  AN ERROR IS RETURNED IF   */
/*  NEGATIVE DATA IS ENCOUNTERED DURING A LOG SCALE OR IF AN     */
/*  INVALID PLOT TYPE CODE IS RECEIVED.                          */
/*                                                                */
/*----------------------------------------------------------------*/

    switch (plot_type)
      {
      case  LINEAR:
        p_linsc(x_data ,num_pts ,SPACING ,X_AXIS_LEN ,TS_ADJ,
            DES_TICSP ,&first_x ,&delta_x ,&x_tic_space);

        p_linsc(y_data ,num_pts ,SPACING ,Y_AXIS_LEN ,TS_ADJ,
            DES_TICSP ,&first_y ,&delta_y ,&y_tic_space);

        p_axis(X_ORIG ,Y_ORIG ,x_axis_title ,x_axis_units,
            TL_CW ,DP_AUTO ,LS_NO ,TIC_CW ,TS_NO,ST_NO,
            X_AXIS_LEN ,X_ANGLE ,first_x ,delta_x,
            x_tic_space);

        p_axis(X_ORIG ,Y_ORIG ,y_axis_title ,y_axis_units,
            TL_CCW ,DP_AUTO ,LS_FIR ,TIC_CCW ,TS_NO ,ST_YES,
            Y_AXIS_LEN ,Y_ANGLE ,first_y ,delta_y,
            y_tic_space);

        p_linlin(num_pts ,x_data ,SPACING ,first_x ,delta_x,
            x_tic_space ,y_data ,SPACING ,first_y ,delta_y,
            y_tic_space , SYM_CNTL ,SYMBOL);

        break;

      case LOG_LINEAR:
        p_linsc(x_data ,num_pts ,SPACING ,X_AXIS_LEN ,TS_ADJ,
            DES_TICSP , &first_x , &delta_x , &x_tic_space);

        if (NORMAL != p_logsc(y_data ,num_pts ,SPACING ,
                Y_AXIS_LEN ,TS_ADJ ,&first_y ,&y_tic_space))
```

```
         {
         printf("\n Bad data for LOG_LINEAR");
         return(ERROR);
         };

     p_axis(X_ORIG ,Y_ORIG ,x_axis_title ,x_axis_units,
        TL_CW ,DP_AUTO ,LS_NO ,TIC_CW, TS_NO ,ST_YES ,
        X_AXIS_LEN, X_ANGLE ,first_x , delta_x ,x_tic_space);

     p_lgaxs(X_ORIG ,Y_ORIG ,y_axis_title ,y_axis_units,
        TL_CCW ,LS_FIR ,TIC_CCW ,TS_NO ,ST_YES ,Y_AXIS_LEN,
        Y_ANGLE ,first_y ,y_tic_space);

     p_loglin(num_pts ,x_data ,SPACING ,first_x ,delta_x,
        x_tic_space ,y_data ,SPACING ,first_y ,y_tic_space,
        SYM_CNTL ,SYMBOL);


     break;

  case LINEAR_LOG:
     if (NORMAL != p_logsc(x_data ,num_pts ,SPACING,
           X_AXIS_LEN ,TS_ADJ ,&first_x ,&x_tic_space))
        {
        printf("\n Bad data for LINEAR_LOG");
        return(ERROR);
        };

     p_linsc(y_data ,num_pts ,SPACING ,Y_AXIS_LEN ,TS_ADJ,
        DES_TICSP ,&first_y ,&delta_y ,&y_tic_space);

     p_lgaxs(X_ORIG ,Y_ORIG ,x_axis_title ,x_axis_units,
        TL_CW ,LS_NO ,TIC_CW ,TS_NO ,ST_YES ,X_AXIS_LEN,
        X_ANGLE ,first_x ,x_tic_space);

     p_axis(X_ORIG ,Y_ORIG ,y_axis_title ,y_axis_units,
        TL_CCW ,DP_AUTO ,LS_FIR ,TIC_CCW ,TS_NO ,ST_YES,
        Y_AXIS_LEN ,Y_ANGLE ,first_y ,delta_y,
        y_tic_space);

     p_linlog(num_pts ,x_data ,SPACING ,first_x ,x_tic_space,
        y_data ,SPACING ,first_y ,delta_y ,y_tic_space,
        SYM_CNTL ,SYMBOL);

     break;

  case LOG_LOG:
     if (NORMAL != p_logsc(x_data ,num_pts ,SPACING,
           X_AXIS_LEN ,TS_ADJ ,&first_x ,&x_tic_space))
        {
        printf("\n Data error X LOG_LOG");
        return(ERROR);
        };

     if (NORMAL != p_logsc(y_data ,num_pts ,SPACING,
              Y_AXIS_LEN ,TS_ADJ ,&first_y ,&y_tic_space))
             {
             printf("\n Data error Y LOG_LOG");
             return (ERROR);
             };

          p_lgaxs(X_ORIG ,Y_ORIG ,x_axis_title ,x_axis_units,
             TL_CW ,LS_NO ,TIC_CW ,TS_NO ,ST_YES ,X_AXIS_LEN,
             X_ANGLE ,first_x ,x_tic_space);

          p_lgaxs(X_ORIG ,Y_ORIG ,y_axis_title ,y_axis_units,
             TL_CCW ,LS_FIR ,TIC_CCW ,TS_NO ,ST_YES ,Y_AXIS_LEN,
             Y_ANGLE ,first_y ,y_tic_space);

          p_loglog(num_pts ,x_data ,SPACING ,first_x ,x_tic_space,
             y_data ,SPACING ,first_y ,y_tic_space,
             SYM_CNTL ,SYMBOL);

             break;

        default:
          {
          printf("\n Plot type select error");
          return (ERROR);
          };
     }




/*----------------------------------------------------------------*/
/*                                                                */
/*  THIS BLOCK PLOTS THE TITLE OF THE GRAPH AT THE TOP OF THE     */
/*  GRAPH AND CLOSES THE PLOTTING DEVICE.  IT THEN RETURNS TO     */
/*  CALLING PROGRAM.                                              */
/*                                                                */
/*----------------------------------------------------------------*/

  p_pltncr(TITLE_X ,TITLE_Y ,UP);
  p_stchr(CHR_WIDTH ,CHR_HEIGHT ,CHR_ANGLE);
  p_textc(plot_title);
  p_closp();
  return (NORMAL);
}
```

SIMULATION OF NON–IDEALITIES IN FREQUENCY COMPRESSIVE RECEIVER COMPONENTS AND INPUTS

by

PHILLIP EDWARD FRY

B.S., Kansas State University, 1987

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

ELECTRICAL ENGINEERING
KANSAS STATE UNIVERSITY
Manhattan, Kansas
1988

ABSTRACT

This thesis presents the results of computer simulations performed on a frequency compressive receiver. The simulation introduced non–idealities into receiver components and inputs one at a time to determine the effect upon performance.

Non–ideal models for the sweeping local oscillator and the dispersive filter were implemented. Linear and non–linear errors were introduced into these component's frequency vs. time relationships. All errors resulted in a reduced probability of detection and reduced spectral resolution. All but the linear dispersive filter error resulted in frequency determination errors. Non–linear dispersive filter errors had the least detrimental effect on performance.

Three classes of non–ideal (ideal is a long CW pulse) inputs were modeled: short CW pulses, long FSK pulses and long PSK pulses. Short pulses resulted in low probability of detection and reduced spectral resolution. FSK input pulses, depending on the frequency and timing, can have greatly reduced energy per output peak. Therefore, a potential problem exists in detecting all the frequencies used. PSK input pulses that undergo a phase transition during the sweep resulted in a notch in the output pulse when the timing was right. A mathematical derivation was used to explain this effect.