To the Graduate Council:

I am submitting herewith a thesis written by H. David Helsley, Jr. entitled "A Study of Zero-Input Limit Cycles in Floating-Point Digital Signal Processors." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Bruce W. Bomar

Dr. Bruce W. Bomar, Major Professor

We have read this thesis and recommend its acceptance:

L Montgomery Smith

Roy S Joseph

Accepted for the Council:

Swinkel

Associate Vice Chancellor
and Dean of the Graduate School

A Study of Zero-Input Limit Cycles in Floating-Point Digital Signal Processors

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Harold David Helsley, Jr.
May, 1995

To my wonderful wife,

Teressa Helsley

and to my parents,

Elaine and Jack Hagerman,

and David and Dorothy Helsley.

Without all of your prayers and support,

this thesis

and my degree

would not be possible.

Acknowledgments

The author would like to thank Dr. Bruce W. Bomar, Dr. Roy D. Joseph, and Dr. L. Montgomery Smith for serving on my Master's Thesis committee and for providing me with suggestions which have greatly improved this thesis. A very special thank-you goes to Dr. Bomar, my major professor, who provided many suggestions when I wasn't sure how to proceed, who encouraged me when I needed it, and whose insight has been nothing short of inspirational.

Abstract

An experimental analysis of zero-input limit cycles in stable, second-order infinite impulse response filters with complex-conjugate poles implemented on the Texas Instruments TMS320C31 floating-point digital signal processor was performed. Results presented are based on studies using three second-order filter realizations. Limit cycle behavior was experimentally observed by moving transfer function poles across a dense grid within the unit circle in the z-plane, providing a complete study for the Direct Form, Minimum Roundoff Noise and Type III realizations.

Limit cycles were found to be negligibly small for all three realizations, and their existence was found to be dependent upon the filter realization and the locations of the poles. In all three realizations, higher-Q filters (with pole radii close to the unit circle) were found to be more susceptible to limit cycles than lower-Q filters. Although transfer function poles were moved out to radii of 0.9999 for each filter realization, limit cycle magnitudes remained very small (around 10^{-37}). Cases involving rounding, truncation and magnitude-truncation of the state variables were investigated. The results were found to be virtually identical for all three indicating another source of the limit cycles.

A detailed analysis of the effects of the processor's underflow flush-to-zero characteristic was performed. Deviations from the expected behavior of the filter were found to be correlated with instances where, while updating the state variables, the results of one or more calculations were "flushed" to zero. This final result suggests that the underflow flush-to-zero characteristic of the processor is the primary cause of the observed limit cycles.

Table of Contents

List of Figures

Chapter I - Introduction

Today, floating-point digital signal processors are used in real-time filtering applications for several reasons: they are less expensive (Texas Instruments has recently advertised its TMS320C32 DSP for under $10 in volume quantities), the floating-point DSP can represent a vastly wider range of numbers with increased precision, and they tend to be less susceptible to quantization effects than fixed-point processors. In the past, when fixed-point processors were widely used, many authors worked theoretically and experimentally to analyze the limit-cycle characteristics of digital filters implemented with fixed-point arithmetic [1-6]. More recently, theoretical work by Bauer [7] and Kaneko [8] shed light on the limit-cycle characteristics of floating-point digital filters. However, they do not account for the case where underflow levels are automatically flushed to zero (which is a characteristic of many commercially available DSPs). It is not possible for a digital filter to operate in the "underflow regime" as is assumed in [7] when DSPs with underflow flush-to-zero are used.

Limit cycles are parasitic oscillations caused by quantization introduced in the recursive part of any infinite impulse response (IIR) filter. Since the result of a floating-point arithmetic operation produces more bits of precision than the processor can represent, some method of quantization is necessary, and this quantization nonlinearity may produce low-level oscillations in the presence of a sustained zero input, even though the filter is a stable linear system. Also, on DSPs, if the result of any arithmetic operation is too small to be represented, that result will be flushed to zero, and this introduces another nonlinearity. Even though floating-point processors like the Texas Instruments TMS320C31 have so-called "extended-precision registers", they do not carry enough bits of precision to hold the exact result of a floating-point multiply or add, and all extended-

precision numbers must be quantized to single precision format before they can be used [11].

The problem of limit cycles, it seems, is here to stay. Bauer [7] and Kaneko [8] both agree that limit cycles can exist for digital filters realized with floating-point arithmetic. However, there has not been any work which attempts to analyze exactly when limit cycles will occur and what their amplitudes will be in a floating-point DSP with extended-precision intermediate results and the underflow flush-to-zero characteristic. It would be useful to know whether a given filter will produce zero-input limit cycles based solely on the characteristics of the filter. The work presented in this thesis addressed this question by experimentally searching for and recording the magnitude of limit cycles for a typical second-order section through a dense grid of possible pole locations in the z-plane. The work presented is an experimental measurement of limit cycle amplitudes for three different realizations of causal, stable, second-order IIR filters with complex-conjugate poles. These realizations are the Direct Form realization [13] (which is computationally efficient, but has poor roundoff noise characteristics), the Minimum-Roundoff-Noise realization [9] (which has good roundoff noise characteristics, but requires much more computation to implement), and what is called the Type III realization [9] (which represents a compromise between being computationally efficient and having good roundoff noise characteristics).

Three common methods of quantizing the state variables were used: rounding, truncation, and magnitude truncation. If the result of an arithmetic operation is half-way between two quantization levels or higher, rounding pushes that result up to the next higher quantization level; otherwise, rounding drops its value to the next lower quantization level. Truncation simply removes any extra bits of precision on the mantissa above the maximum number as dictated by the architecture of the processor. In the C31,

any mantissa bits to the right of the twenty-fourth bit are removed. Truncation always decreases the value of the floating-point number (positive or negative). Magnitude truncation produces the truncated value of positive numbers. For negative numbers, however, the number is first negated (making it positive), then it is truncated, and then the number is negated again. Magnitude truncation, therefore always decreases the magnitude of the number. Regardless of the quantization method used, however, the experimental results observed were virtually the same. This result led to a study of the influence of underflow flush-to-zero on the limit cycles and to the conclusion that the underflow flush-to-zero characteristic of the DSP was the dominant cause of the observed limit cycles.

The thesis is divided as follows: Chapter II provides some background information on limit cycles, state-space structures, the filter transfer function and the three second-order filter realizations which were considered, the Texas Instruments TMS320C31 digital signal processor, and the generalized norm (which was used to analyze the effects of underflow flush-to-zero). Chapter III describes how the search for limit cycles was accomplished and what other avenues were explored. Chapter IV presents the results of the procedures outlined in Chapter III along with a short discussion about the results, and Chapter V provides a summary of the thesis.

Chapter II - Background

This chapter provides the reader with the necessary background for understanding the remainder of this thesis. First, limit cycles are discussed in detail, and an example of a limit cycle in a second-order filter is presented. Next, the transfer function of the filters which are implemented in the experiments is derived. Then, each of the second-order filters are realized in state-space form. After that, some facts about the Texas Instruments C31 DSP are presented. Finally, the generalized norm is introduced in preparation for the detailed analysis of the effects of underflow flush-to-zero on the impulse response.

Limit Cycles

Finite precision arithmetic causes limit cycles. In any digital signal processor, a finite number of bits is used to represent any number, be it fixed- or floating-point. Fixed-point addition produces a sum which can be represented by the same number of bits as its operands; however, the result of a floating-point addition usually requires more bits to represent than either of its operands. Both fixed- and floating-point multiplication generate products which require more bits to represent than either of their operands. Therefore, some method of quantization is necessary to return the resulting sum or product to normal precision. Any quantization introduced into a stable, linear, shift-invariant system can render the system unstable, since the nonlinearities due to quantization make the overall system nonlinear. One possible result of quantization nonlinearities is a limit cycle.

Since this thesis considers limit cycles in detail, an example is in order. To keep the example simple, fixed-point arithmetic is used, although the effect for floating-point processors is similar. Consider a simple, stable, second-order system described by the difference equation

$$y(n) = (3/4)\, y(n-1) - (7/8)\, y(n-2) + x(n) \tag{2.1}$$

Assume that this filter is realized with a fixed-point processor which carries three bits of precision and a sign bit to the left of the binary decimal point. Representable numbers therefore range from -1 (1.000) to +7/8 (0.111) in step sizes of 1/8. Since y(n) must also be representable by the processor, the result of each filter iteration can be written as

$$y(n) = Q_r[\, (3/4)\, y(n-1) - (7/8)\, y(n-2) + x(n)\,] \tag{2.2}$$

where $Q_r[\cdot]$ is defined as the result of quantization due to rounding. Rounding was arbitrarily chosen as the quantization method because it is well-known and easy to understand. Assume that the entire multiply-add sequence is done in full precision (stored in a 7-bit accumulator, say), and then quantized. Assume also that y(-1) = y(-2) = 0 as initial conditions and the input sequence is x(n) = {5/8, 1/8, 0, 0, ...}. The filter produces the following output sequence:

$$\begin{aligned}
y(0) &= Q_r[(3/4)\,(0) - (7/8)\,(0) + 5/8] = 5/8 \\
y(1) &= Q_r[(3/4)\,(5/8) - (7/8)\,(0) + 1/8] = Q_r[19/32] = Q_r[4.75/8] = 5/8 \\
y(2) &= Q_r[(3/4)\,(5/8) - (7/8)\,(5/8) + 0] = Q_r[-5/64] = -1/8 \\
y(3) &= Q_r[(3/4)\,(-1/8) - (7/8)\,(5/8) + 0] = Q_r[-41/64] = -5/8 \\
y(4) &= Q_r[(3/4)\,(-5/8) - (7/8)\,(-1/8) + 0] = Q_r[37/64] = 5/8 \\
y(5) &= Q_r[-5/64] = -1/8 \\
y(6) &= Q_r[-41/64] = -5/8 \\
y(7) &= Q_r[37/64] = 5/8 \\
y(8) &= Q_r[-5/64] = -1/8 \\
y(9) &= Q_r[-41/64] = -5/8 \\
y(10) &= Q_r[37/64] = 5/8 \\
y(11) &= Q_r[-5/64] = -1/8 \\
y(12) &= Q_r[-41/64] = -5/8
\end{aligned} \tag{2.3}$$

Notice that the repetitive nature of the limit cycle prevents the output from ever reaching zero, even though it is a stable filter.

One way of reducing the seriousness of limit cycles is to increase the precision with which numbers are represented. This reduces the step size between representable numbers (which was 1/8 in this example) and as a result, reduces the amplitude of any

limit cycles. Another less attractive method is to change the filter which is being realized. The example filter had complex-conjugate poles at a radius of 0.935 and an angle of 62 degrees. If one or both of the coefficients had been reduced (especially the coefficient of y(n-2), which is the square of the radius), the output would have been driven to small enough numbers to round to zero, pushing the state variables to zero and eliminating the limit cycle. Again, though, this involves changing the characteristics of the filter being implemented, and this is probably an unacceptable alternative.

Another possibility is to change the filter realization. If a state-space realization were used, it might produce better results. It has been shown that some state-space realizations are less susceptible to finite wordlength effects [2].

The Transfer Function

To perform an exhaustive search for limit cycles, it was necessary to restrict the filter transfer function which was studied. First, only IIR filters which are causal and stable were considered. This is justifiable because in the practice of designing real-time DSP systems, only causal and stable filters are used. Next, only the important case of second-order filters was considered, since any higher order filter can be decomposed into first- and second-order subfilters in either a cascade or parallel connection. Finally, only those filters which have complex-conjugate poles were considered, since real poles can be studied as first-order sections, and complex poles which are not complex-conjugates of one another would require complex filter coefficients.

These restrictions significantly narrowed the search parameters. Since only causal and stable filters were considered, the poles of the transfer function were restricted to be inside the unit circle in the z-plane. Further, these poles must be complex-conjugates of one another. The zeroes of the transfer function do not affect limit-cycle behavior, because they have no bearing on the recursive part of the filter. Since the placement of the

zeros is arbitrary, the transfer function studied was:

$$H(z) = \frac{z}{(z - r e^{j\theta})(z - r e^{-j\theta})} \tag{2.4}$$

Note that the poles of this transfer function conform to all restrictions if $0 < r < 1$ and $0 < \theta < 180°$. This particular transfer function's numerator was chosen so that the inverse z-transform or impulse response could be found in a closed form. Rewriting the denominator produces:

$$H(z) = \frac{z}{z^2 - 2 r \cos(\theta) z + r^2} \tag{2.5}$$

whose inverse z-transform is:

$$h(n) = r^{n-1} \frac{\sin(\theta n)}{\sin(\theta)} u(n) \tag{2.6}$$

From (2.5), it is evident that all of the coefficients are real. Also, if the entire space within the unit circle is to be accounted for by this transfer function, it is required that the pole radius, r, be varied from a value close to zero to a value very close to (but not equal to) 1, and that the pole angle, θ, be varied from a value close to 0 to a value close to 180°.

Equation (2.6) can be manipulated to generate some informative results concerning the decay of h(n). Let us assume that we want h(n) to decay to an absolute value which is smaller than some non-negative threshold, T, and we need to know how long this will take. This is useful in determining when h(n) has decayed below the smallest representable number in the DSP, and hence, when limit cycles are present. So, setting the absolute value of the impulse response in (2.6) less than or equal to the threshold T yields

$$T \geq |r^{n-1}| \frac{|\sin(\theta n)|}{|\sin(\theta)|} u(n). \tag{2.7}$$

If $0 < r < 1$, $0 < \theta < 180°$, and $n \geq 0$, then

$$T \sin(\theta) \geq r^{n-1} |\sin(\theta n)| \tag{2.8}$$

The right side of (2.8) is less than or equal to r^{n-1}, and therefore the $|\sin(\theta \cdot n)|$ term can be dropped. Rearranging and taking the logarithm of both sides then produces

$$(n-1) \log(r) \leq \log(T \sin(\theta)) \tag{2.9}$$

and therefore,

$$n \geq 1 + \frac{\log(T \sin(\theta))}{\log(r)} \tag{2.10}$$

This equation gives n for which the impulse response magnitude has decayed below the threshold, T, in the absence of a limit cycle with amplitude T or greater. This is used in determining when the normal impulse response magnitude oscillations drop below the smallest representable floating-point number that the processor can handle, and hence, when the limit cycle oscillations are clearly observable.

State-Space Representation of Filters

A popular method of realizing recursive filters uses the state-space representation:

$$\underline{\mathbf{s}}[n+1] = \mathbf{A}\, \underline{\mathbf{s}}[n] + \underline{\mathbf{b}}\, x[n] \tag{2.11a}$$
$$y[n] = \underline{\mathbf{c}}^{t}\, \underline{\mathbf{s}}[n] + d\, x[n] \tag{2.11b}$$

Here, vectors are represented by boldface and underlining, matrices are represented by boldface, and scalar values are represented by normal text. The first equation updates the state vector $\underline{\mathbf{s}}[n]$ as a function of the current value of $\underline{\mathbf{s}}[n]$ and the input, x[n]. The second equation determines the output as a function of the current values of the state vector and the input.

Taking the z-transform of (2.11) produces

$$z\,\underline{\mathbf{S}}(z) = \mathbf{A}\,\underline{\mathbf{S}}(z) + \underline{\mathbf{b}}\,X(z) \tag{2.12a}$$
$$Y(z) = \underline{\mathbf{c}}^{t}\,\underline{\mathbf{S}}(z) + d\,X(z) \tag{2.12b}$$

Solving for $\underline{\mathbf{S}}(z)$ in (2.12a) and substituting into (2.12b) yields

$$Y(z) = \underline{\mathbf{c}}^{t}\,(z\mathbf{I} - \mathbf{A})^{-1}\,\underline{\mathbf{b}}\,X(z) + d\,X(z) \tag{2.13}$$

The transfer function of a filter in state-space form is, then

$$H(z) = Y(z)/X(z) = \underline{\mathbf{c}}^{t}\,(z\mathbf{I} - \mathbf{A})^{-1}\,\underline{\mathbf{b}} + d \tag{2.14}$$

For a second-order system, $\underline{\mathbf{s}}$, and $\underline{\mathbf{b}}$, are 2-by-1 column vectors, $\underline{\mathbf{c}}^{t}$ is a 1-by-2 row vector, and $\mathbf{A}$ is a 2-by-2 matrix. In this case, (2.11) can be rewritten as:

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} x[n] \tag{2.15a}$$

$$y[n] = \begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + d\,x[n] \tag{2.15b}$$

An illustration of the second-order state-space filter which uses the coefficients defined in (2.15) is shown in Figure (2.1).

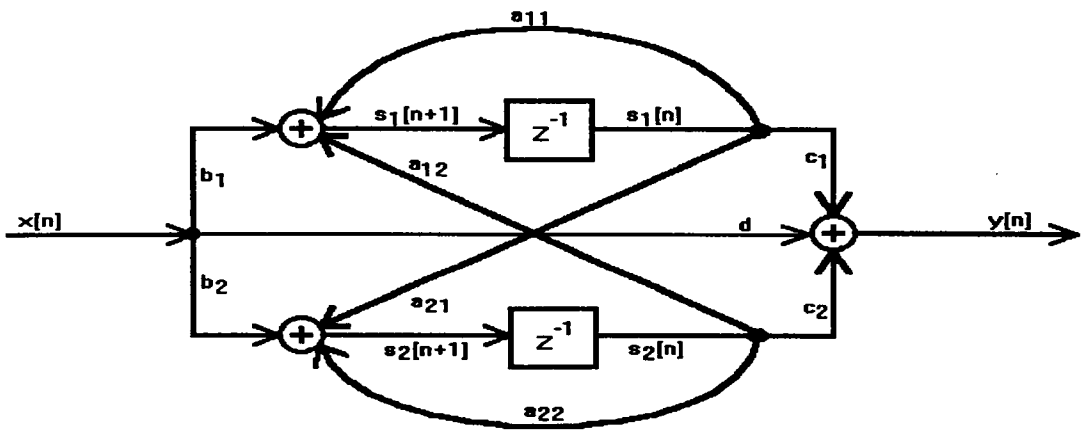

Figure (2.1) - State Space Representation

Three different state-space realizations are now presented for the transfer function of (2.5). An infinite number of additional realizations can be found, however, via a linear transformation of the state vector.

Direct Form Realization

For the general second-order transfer function

$$H(z) = \frac{z\,\alpha_1 + \alpha_2}{z^2 + z\,\beta_1 + \beta_2} \tag{2.16}$$

the Direct-Form Realization written in state-space form is

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} -\beta_1 & -\beta_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x[n] \tag{2.17a}$$

$$y[n] = [\alpha_1 \quad \alpha_2] \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} \tag{2.17b}$$

Using (2.16) and (2.17), the general transfer function in (2.5) is

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} 2\,r\cos(\theta) & -r^2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} x[n] \tag{2.18a}$$

$$y[n] = [\,1 \quad 0\,] \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} \tag{2.18b}$$

Note that for this particular realization, only one state variable is calculated and quantized for each iteration. The updated state variable $s_2[n+1]$ is simply a copy of whatever $s_1[n]$ is. This realization does have the advantage of being computationally efficient and easy to implement, but it has poor roundoff noise characteristics [9]. For the filter shown in (2.18), the Direct Form realization requires only two multiplies and two adds for each filter iteration. The Direct-Form realization as a state-space structure is illustrated in Figure (2.2) for the transfer function of (2.5).

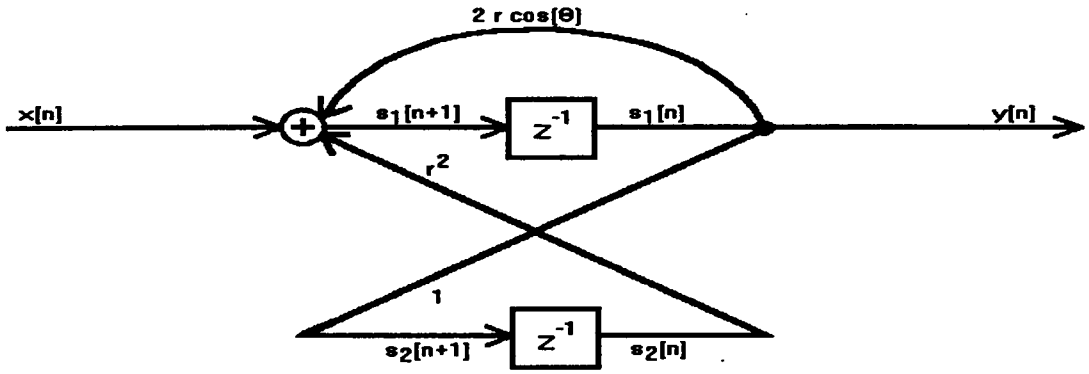

Figure (2.2) - Direct Form Realization

Minimum-Noise State-Space Realization

The Minimum-(Roundoff)-Noise second-order state-space realization was analyzed by Jackson, Lindgren, and Kim [10]. The general transfer function in (2.16) compared with the expressions in (2.14) and (2.15) produces the following four constraining equations [9]:

$$\alpha_1 = c_1 b_1 + c_2 b_2 \quad (2.19a)$$
$$\alpha_2 = c_1 b_2 a_{12} + c_2 b_1 a_{21} - c_1 b_1 a_{22} - c_2 b_2 a_{11} \quad (2.19b)$$
$$\beta_1 = -(a_{11} + a_{22}) \quad (2.19c)$$
$$\beta_2 = a_{11} a_{22} - a_{12} a_{21} \quad (2.19d)$$

The Minimum-Noise structure is further constrained by requiring [10]

$$a_{11} = a_{22} = \varphi \quad (2.20a)$$
$$b_1 c_1 = b_2 c_2 = \lambda \quad (2.20b)$$

When the transfer function of interest in (2.5) is considered, using (2.19) and (2.20) yields the following

$$1 = 2\lambda \tag{2.21a}$$
$$2\lambda\varphi = c_1 b_2 a_{12} + c_2 b_1 a_{21} \tag{2.21b}$$
$$r\cos(\theta) = \varphi \tag{2.21c}$$
$$r^2 = a_{11} a_{22} - a_{12} a_{21} \tag{2.21d}$$

Since the filter was implemented on a floating-point DSP, numbers larger than +1 can be easily represented and it is not necessary to scale the realization. One choice of filter parameters which meet the requirements in (2.21) is

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} r\cos(\theta) & r\sin(\theta) \\ -r\sin(\theta) & r\cos(\theta) \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} \sin(\theta) \\ \cos(\theta)+1 \end{bmatrix} x[n] \tag{2.22a}$$

$$y[n] = \begin{bmatrix} (2\sin(\theta))^{-1} & (2\cos(\theta)+1)^{-1} \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} \tag{2.22b}$$

However, a linear transformation of the state vector given by

$$\underline{s}' = \mathbf{T}^{-1}\underline{s} \tag{2.23}$$

where $\mathbf{T}$ is any nonsingular 2-by-2 matrix, produces a more computationally efficient realization. This linear transformation generates a new state-space realization

$$\begin{aligned} \mathbf{A}' &= \mathbf{T}^{-1}\,\mathbf{A}\,\mathbf{T} \\ \underline{b}' &= \mathbf{T}^{-1}\,\underline{b} \\ \underline{c}' &= \mathbf{T}^{t}\,\underline{c} \\ d' &= d \end{aligned} \tag{2.24}$$

Using the transformation matrix,

$$\mathbf{T} = \begin{bmatrix} \sin(\theta) & 0 \\ 0 & \cos(\theta)+1 \end{bmatrix} \tag{2.25}$$

the following transformed filter realization is produced:

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} r\cos(\theta) & r(\cos(\theta)+1) \\ -r\sin^2(\theta)/(\cos(\theta)+1) & r\cos(\theta) \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} x[n] \tag{2.26a}$$

$$2\,y[n] = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} \tag{2.26b}$$

Notice that this realization also satisfies the constraints in (2.19) and (2.20). In (2.26b), a constant term of 1/2 was factored out of the $\underline{c}^t$ row vector and a factor of 2 was multiplied at the output. In essence, 2 y[n] is realized, but then it can be divided by two to produce y[n]. When this scheme is used, a total of five multiplies and five adds are required to perform each filter iteration. This realization requires much more computation than the Direct Form presented earlier, but it is known that this realization is less susceptible to finite wordlength effects [9]. The Minimum-Noise structure is illustrated in Figure (2.3).

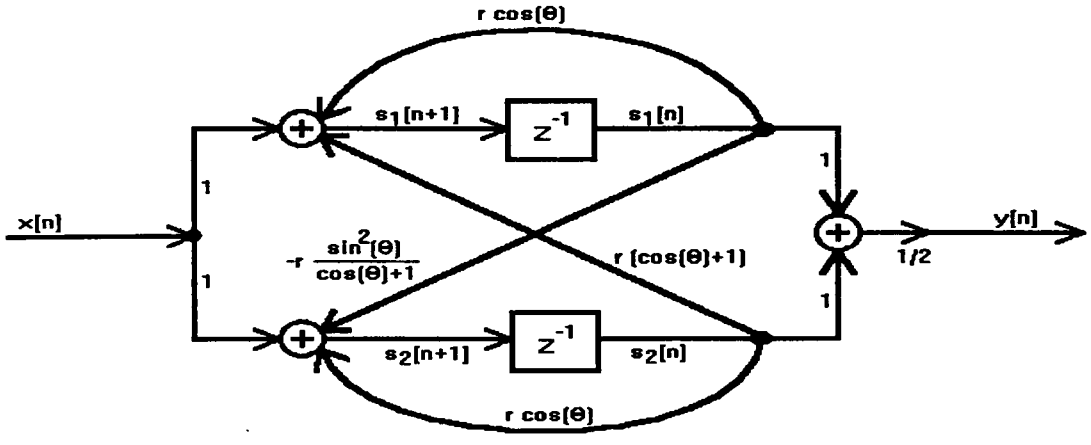

Figure (2.3) - Minimum-Noise Realization

Type III Realization

The third and final filter realization which will be considered is the Type III realization presented in [9]. For the general second-order filter in (2.16), the Type III realization is:

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} -\beta_1/2 & 1 \\ (\beta_1/2)^2-\beta_2 & -\beta_1/2 \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x[n] \qquad (2.27a)$$

$$y[n] = [\,\alpha_2-(\alpha_1\,\beta_1/2)\quad \alpha_1\,] \begin{bmatrix} s1[n] \\ s2[n] \end{bmatrix} \tag{2.27b}$$

When the transfer function of (2.5) is considered the following structure results:

$$\begin{bmatrix} s_1[n+1] \\ s_2[n+1] \end{bmatrix} = \begin{bmatrix} r\cos(\theta) & 1 \\ -(r\sin(\theta))^2 & r\cos(\theta) \end{bmatrix} \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x[n] \tag{2.28a}$$

$$y[n] = [\,r\cos(\theta) \quad 1\,] \begin{bmatrix} s_1[n] \\ s_2[n] \end{bmatrix} \tag{2.28b}$$

This realization is not minimum-noise, since it violates (2.20b), but it does satisfy (2.20a). The advantage of using a Type III Realization is that it requires less computation than the Minimum-Noise structure - only three multiplies (notice that $y[n] = s_1[n+1]$ here) and three adds for this special case. In general, there may be more operations (depending on the values of α_1, α_2, β_1, and β_2), but it still requires fewer computations than the Minimum-Noise structure. The Type III structure is "almost as good" as the Minimum-Noise structure in reducing roundoff noise, and it is almost as efficient to realize as the Direct Form for this special case. It is a good compromise between noise-immunity and computational efficiency. The Type III structure is shown in Figure (2.4).

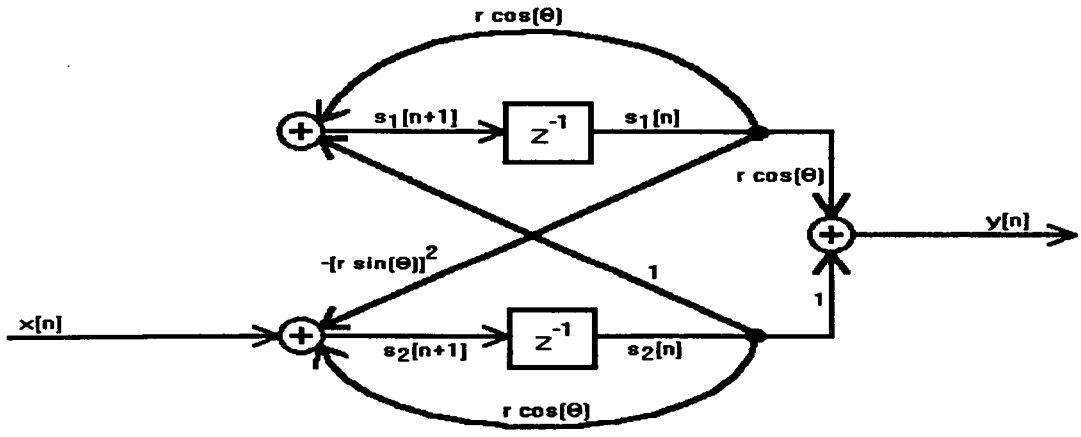

Figure (2.4) - Type III Realization

The C31 Floating-Point Digital Signal Processor

The Texas Instruments C31 DSP is a 32-bit floating-point processor whose 40-MHz version is capable of performing 40 million floating-point operations per second. Each single-precision floating-point number carries 24 mantissa bits (which includes a sign bit) and an 8-bit exponent. Eight internal registers carry an extra 8 bits of mantissa precision (called extended-precision registers).

The C31, like other floating-point DSPs, has a characteristic known as "Underflow Flush-to-Zero". Whenever the result of an arithmetic operation is calculated to be less than the smallest number representable in the DSP in normalized form (which is $\pm 5.877 \times 10^{-39}$ for single precision [11]), the processor automatically "flushes" that value to zero. This prevents the processor from operating in an underflow regime with unnormalized floating-point numbers and adds an extra nonlinearity to the problem of theoretically analyzing limit cycles in this DSP. This characteristic, however, does not automatically "force" the filter's output to zero, because while one state variable may be "flushed" to zero, the other one may still be representable by the processor. In the Direct-Form Realization, for example, the final calculation in generating the first state variable must be "flushed" to zero twice in a row in order to stop the oscillations, since the second state variable is just a delayed version of the first.

The Generalized Norm

Mills, Mullis and Roberts [12] provide a set of conditions which, when met, precludes the existence of overflow oscillations and zero-input limit cycles in fixed-point filter implementations that use magnitude truncation quantization. To begin, note that if the input to the filter is zero, then the state update equation is

$$\underline{s}(n+1) = Q_m[\, \mathbf{A}\, \underline{s}(n)\,] \tag{2.29}$$

where $Q_m[\cdot]$ represents the result after magnitude truncation. In order to eliminate limit

cycles, $\underline{s}(n)$ must eventually decay to zero. A sufficient condition which will lead to this is that there exist some positive real numbers d_1^2 and d_2^2 for which

$$\| \underline{s}(n+1) \| < \| \underline{s}(n) \| \tag{2.30}$$

where the generalized length function (norm) is defined as

$$\| \underline{s}(n) \| = \sqrt{[d_1^2 \, s_1^2(n) + d_2^2 \, x_2^2(n)]} \tag{2.31}$$

If the generalized length of the state vector decreases with each recursion of (2.29), eventually the state variables which make up $\underline{s}(n)$ also decay to zero, precluding the possibility of a limit cycle. Since magnitude truncation quantization cannot increase the state variables in (2.29),

$$\| \underline{s}(n+1) \| = \| Q_m[\mathbf{A} \, \underline{s}(n)] \| \leq \| \mathbf{A} \, \underline{s}(n) \| \tag{2.32}$$

In [12], it was shown that for any stable, second-order filter with complex-conjugate poles,

$$\| \mathbf{A} \, \underline{s}(n) \| < \| \underline{s}(n) \| \tag{2.33}$$

as long as

$$\begin{aligned} a_{11} &= a_{22} \\ d_1^2 &= 1 \\ d_2^2 &= | \, a_{12}/a_{21} \, | \end{aligned} \tag{2.34}$$

Therefore, choosing $a_{11} = a_{22}$ ensures that (2.30) is satisfied so that limit cycles cannot exist when magnitude truncation is the only quantization that exists in the filter implementation. This requirement is satisfied in both the Minimum Noise and Type III realizations.

Now that limit cycles have been introduced, the three filter realizations have been given, and some information about the C31 DSP and the generalized norm has been presented, a description of the experimental procedure is given in Chapter III.

Chapter III - Experimental Procedure

Filter Realizations

The three filter realizations (Direct Form, Minimum-Noise, and Type III) were implemented in Texas Instruments assembly language as C-callable subroutines with prototypes which were very similar to one another. Each filter realization required the following five arguments passed to it:

1. A pointer to an array of coefficients needed to realize the filter.
2. The number of recursions for which the filter should be applied. (2.10)
3. A pointer to an array representing the starting state vector $\underline{s}(0)$.
4. A pointer to a similar array where the ending state vector will be stored.
5. The constant input to the filter (usually zero).

An example of one of the function prototypes is:

*float direct_iir(float *coeffs, int iterations, float *start, float *end, float input);*

Notice that the *direct_iir* routine passes back the floating-point value of the output after *iterations* recursions. This function, once written, was then called from a parent program with a combination of filter coefficients, input, starting state vectors, and number of iterations. The reader is referred to the Appendix where the assembly language source code is listed in its entirety for the three different realizations.

In order to check the correctness of these realizations, each routine was called to generate the impulse response of a sample filter, one iteration at a time, storing the output values in an array. The array values were then checked against the anticipated impulse response given in (2.6). A TMS320C31 Emulator was used to watch program execution and debug the routines. When it was certain that the assembly language routines were indeed realizing the filter correctly, they were then used by a master C program which provided the needed arguments in an orderly fashion.

The Search for Limit Cycles

Overview

Once the filtering routines were written, we had three "black boxes" which implemented a given filter using one of three different realizations. They were verified to be correct, and all that was left to do was to use them to search for the limit cycles over the z-plane inside the unit circle. A search algorithm was developed which systematically searches the z-plane and records any limit-cycle activity.

The heart of the search algorithm is a double-nested FOR loop with the pole angle, θ, as a function of the outer loop counter and the pole radius, r, as a function of the inner loop counter. The purpose of this double-nested FOR loop was to systematically cover as much of the z-plane within the confines of the unit circle as possible.

Within each iteration, the required filter coefficients were calculated (based on the values of r and θ), the filter's impulse response was realized out to n=*iterations* with a value of n much greater than the n calculated by (2.10), and then a third loop stepped through a window of 1000 iterations one step at a time, recording the largest output magnitude encountered. Once the window was complete, the largest output in the window was recorded in a large storage array in memory for data retrieval. A flow diagram illustrating the overall program is shown in Figure (3.1).

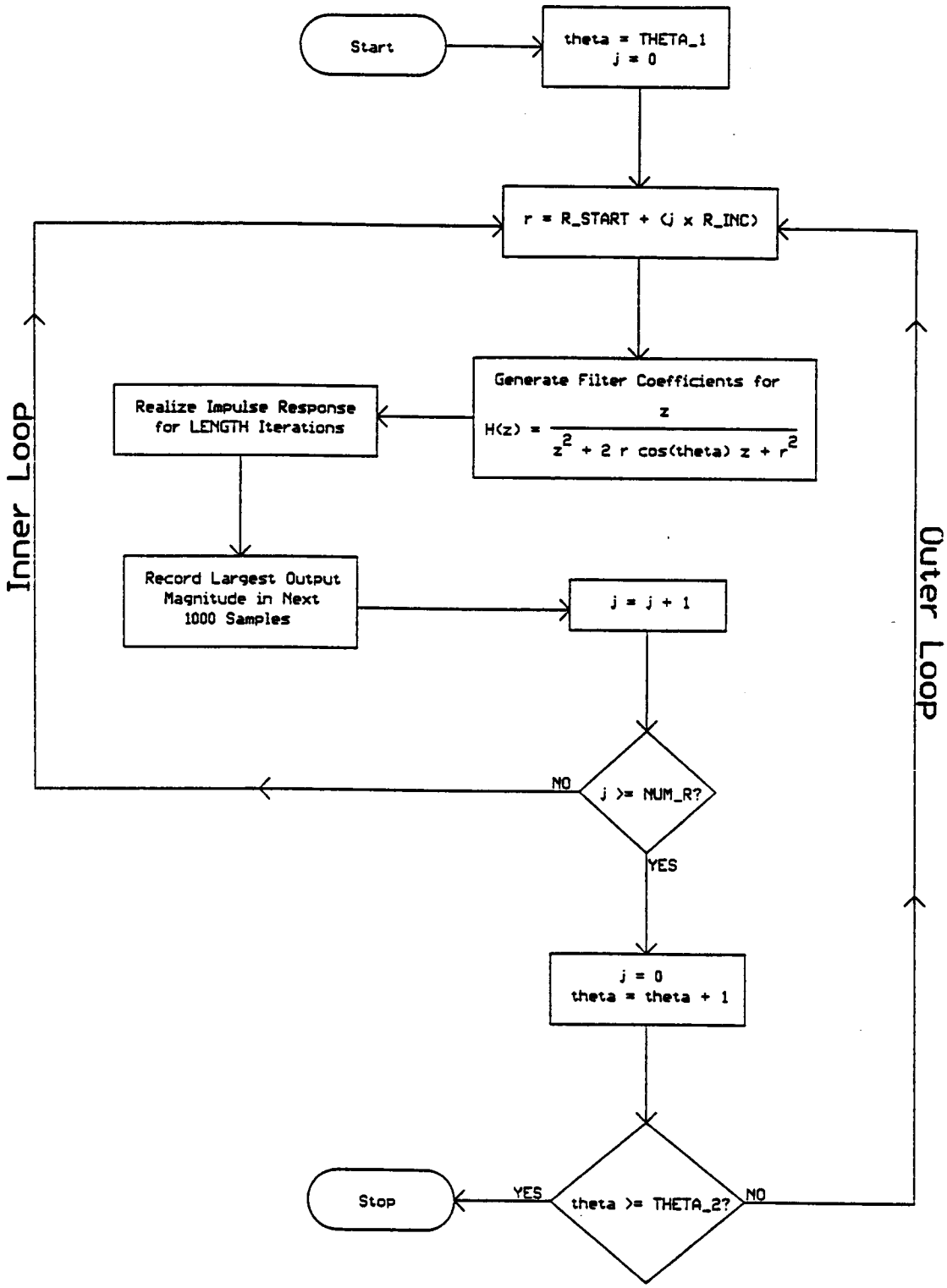

Figure (3.1) - Main Search Algorithm Flow Diagram

Breaking up the Task

The experiments were run on a C31 DSP board which had four independent processors on it. To take advantage of the available hardware, the pole angles were divided into four different sections: 1°-44°, 45°-89°, 90°-134°, and 135°-179°. Each of the four processors worked on exhaustively searching a different quarter of the z-plane. With the problem divided like this, processing time was divided by a factor of four. Preprocessor directives were used in the code to distinguish the starting and ending angles of a program's search. For example, the program "type3a.c" (which realizes the Type III filter for processor A) looked like this:

```
#define   A    1
#include "type3.c"
```

Then, in the main program, a few lines defined the range of angles to sweep through:

```
#ifdef    A
  #define THETA_1   1
  #define THETA_2   45
#endif
```

The outer FOR loop then appears later as:

```
for(i=THETA_1; i<THETA_2; i++)
{   theta = i * PI_OVER_180;
    ...
```

PI_OVER_180 is a constant which converts degrees to radians, which is required of all trigonometric functions called by C (like sine and cosine).

Generating Filter Coefficients

For all three realizations, the filter coefficients depended on the angle and radius of the transfer function's complex-conjugate pole pair. The pole angles were uniformly distributed in 1° increments. Generating the pole radii was accomplished by defining a

minimum starting radius and the number of evenly-spaced radius data desired. The compiler then calculated the correct radius step size. In this way, the limit cycle behavior was examined for pole radii very close to the unit circle without increasing the number of data points beyond a reasonable amount.

For example, we could take ten evenly-spaced data points starting at r=0.5. The radius step size would then be 0.05, and valid radii would then be: {0.5, 0.55, 0.6, ... , 0.9, 0.95}. Notice that the radius does not reach r=1.0. This is because a radius of 1.0 generates a marginally stable filter, and the slightest bit of roundoff noise or coefficient quantization error could make it unstable.

For the example described, the following three pre-processor directives would be used:

```
#define   R_START    0.5
#define   NUM_R      10
#define   R_INC      ((1.0 - (float) R_START) / (float) NUM_R)
```

Notice that the value of *R_INC* is calculated from the values of R_START and NUM_R to be 0.5/10 = 0.05. The above three parameters were used in the inner FOR loop as follows:

```
for(j=0; j<NUM_R; j++)
{    r = R_START + j * R_INC;
     ...
```

In this code, the radius is computed as the sum of the starting radius, R_START, and the product of the loop counter, j, and the radius increment, R_INC.

Once the pole radius and angle were known, coefficients were easily calculated and put into an array of floating-point numbers to pass to the filter realization routine. In the Direct Form, for example, the values of 2 r cos(θ) and $-r^2$ are passed as coefficients. Two simple lines of code in C will easily take care of this.

Calculating the Number of Iterations

Equation (2.10) was used to calculate how many recursions to step through the filter's impulse response before looking for limit-cycle oscillations. For convenience, it is repeated here.

$$n \geq 1 + \frac{\log(T \sin(\theta))}{\log(r)} \qquad (2.10)$$

In this equation, T represents a threshold below which the impulse response magnitude would have decayed after n iterations. The smallest representable number in the C31 digital signal processor is $\pm 5.877 \times 10^{-39}$ for single precision floating point numbers. To be conservative, a magnitude threshold of $T=10^{-40}$ was chosen. If the smallest pole angle, θ, used is 1°, the resulting equation is:

$$n \geq 1 + \frac{\log(10^{-40} \sin(1°))}{\log(r)} \qquad (3.1)$$

or equivalently,

$$n \geq 1 - \frac{41.76}{\log(r)} \qquad (3.2)$$

Values of r which maximize the right side of (3.2) are those which are close to (but not equal to) 1.0. Equation (3.2) was used as a benchmark to determine the minimum number of recursions required to force the filter's impulse response magnitude below the smallest representable number in the DSP. Then, if the impulse response was not zero, a limit cycle must have been present. As long as the value of r used in (3.2) was the closest the poles would get to the unit circle, it would generate a valid number of recursions to step through. For example, with r=0.95 as the closest radius to the unit circle,

$$n_{min} = 1 - \frac{41.76}{\log(0.95)} = 1{,}876 \qquad (3.3)$$

For this case, a conservative choice of n=3,000 might be in order. The author, however, was extremely conservative in the experiments, using n=1,000,000 for radii out to 0.999 (where n_{min} = 96,108), and n=5,000,000 for radii out to 0.9999 (where n_{min} = 961,512).

The Window Loop

Once the filter stepped through the required number of recursions so that the impulse response magnitude should be smaller than the smallest representable value in the DSP, the output of the filter was then checked within a reasonable window length to determine if it was zero or if it was oscillating. If it was oscillating, the largest amplitude of the oscillations within the window was recorded. The following code fragment accomplished this and was done immediately after the initial (long) realization of 1 or 5 million samples:

```
largest = 0.0;
for(n=0; n<1000; n++)
{   output = direct_iir(coeffs, 1, start, start, 0.0);
    if( fabs(output) > largest )
        largest = fabs(output);
}
```

The call to *direct_iir* passes the starting vector in and returns the ending vector to the same array (*start*), allowing the filter assembly language routine to change it after each iteration. After this code was executed, the variable *largest* contained the largest output magnitude encountered in the 1,000-sample window, and it was stored in an array for subsequent data retrieval.

Quantizing the State Variables

Preprocessor directives were used in assembling the filter realization routines to account for different methods of quantizing the state variables after each iteration. The

code is as follows (for the Type III Realization):

```
QUANT .set   1  ; Quantization Method: 1=Round, 2=Truncate,
                ;  3=Magnitude-Truncate, 0=No quantization

;   Realize the filter, calculating s1[n+1] and s2[n+1]
.if      QUANT == 1
   ; Perform Rounding operation on State Variables
.elseif  QUANT == 2
   ; Perform Truncation operation on State Variables
.else
   ; Perform Magnitude-Truncation on State Variables
.endif
;   Go back and realize the filter for the next iteration
```

With the code written in this way, the process of changing state variable quantization methods is simply a matter of changing the value of *QUANT* and recompiling.

Measuring the Limit Cycle Period

An attempt was made to measure the period of the limit cycle by recording the number of filter iterations between identical output blocks. A combination of radius and angle whose filter was experimentally known to support a limit cycle was chosen. The filter was realized for the required number of iterations, and then the next 1000 output samples were recorded in an array, called *array1*. The filter was then stepped through one iteration at a time and the output was compared with the first sample in *array1*. Once an exact match was found, the limit cycle period and the next 999 samples were recorded in a second array, *array2*. A point-by-point comparison of the two output arrays showed whether the limit cycle repeats exactly the same values within that 1000 sample window. The code which determined the limit cycle period is as follows:

```
/* Initial realization */
compare = direct_iir(coeffs, 1.0e+8, start, start, (float) 0.0);
array1[0] = compare;          /* Record first output in array */
for(i=1; i<1000; i++)         /* Record next 999 */
    array1[i] = direct_iir(coeffs, 1, start, start, (float) 0.0);
```

```
/* Find next occurrence of "compare" */
n=1000;                          /* 1000 samples already gone by */
do {    output = direct_iir(coeffs, 1, start, start, (float) 0.0);
        n++;
   } while(output != compare);       /* Step through 1 at a time */

array2[0] = output;                  /* Record output (=compare) */
for(i=1; i<1000; i++)                /* Record next 999 samples */
   array2[i] = direct_iir(coeffs, 1, start, start, (float) 0.0);
```

Changing the Initial State Vector

During the investigation, it was suggested that perhaps the initial conditions of the state vector had some bearing on the limit-cycle characteristics of the filter. This being the case, the impulse response was not an adequate means of measuring the limit-cycle characteristics of the filter. The Direct-Form filter search algorithm was therefore modified to accommodate completely arbitrary initial conditions, and different random initial state vectors were applied to the filter with a zero input.

Tallying the Results

After the filter had been stepped through for all combinations of angles and radius points, the test run was complete and the data was read from the memory of the four DSPs, concatenated together, and put into a readable form for a plotting software package. The results of this effort are given in the next chapter.

Chapter IV - Experimental Results

Exhaustive Search Results

The results of the exhaustive limit-cycle search are now presented. Figures (4.1) to (4.9) are two-dimensional plots with the limit cycle amplitude on the vertical axis, and the pole radius and angle on the horizontal axes. Three parameters vary from plot to plot: the filter realization used, the starting pole radius (and step size), and the number of impulse response samples that were computed before recording the largest output magnitude within a 1,000-sample window. Rounding and truncation were performed on the state variables for the Direct Form and Minimum-Noise structures, and in addition, magnitude truncation was also performed on the Type III realization. The quantization method, however, did not make a noticeable difference, so only the plots for rounding are shown. The plots presented are, however, representative of all three quantization methods. A summary of what each figure represents is given in Table (4.1).

Table (4.1) - List of Figures Showing Limit Cycle Amplitudes

Figure Number	Filter Realization	Radius Range	Initial Number of Samples
(4.1)	Direct Form	0.0 - 0.98	1 Million
(4.2)	Direct Form	0.5 - 0.99	1 Million
(4.3)	Direct Form	0.5 - 0.99	10 Million
(4.4)	Direct Form	0.95 - 0.999	1 Million
(4.5)	Direct Form	0.995 - 0.9999	5 Million
(4.6)	Min.-Noise/Type III	0.5 - 0.99	1 Million
(4.7)	Min.-Noise/Type III	0.95 - 0.999	1 Million
(4.8)	Min.-Noise/Type III	0.995 - 0.9999	5 Million
(4.9)	Min.-Noise/Type III (scaled)	0.995 - 0.9999	5 Million

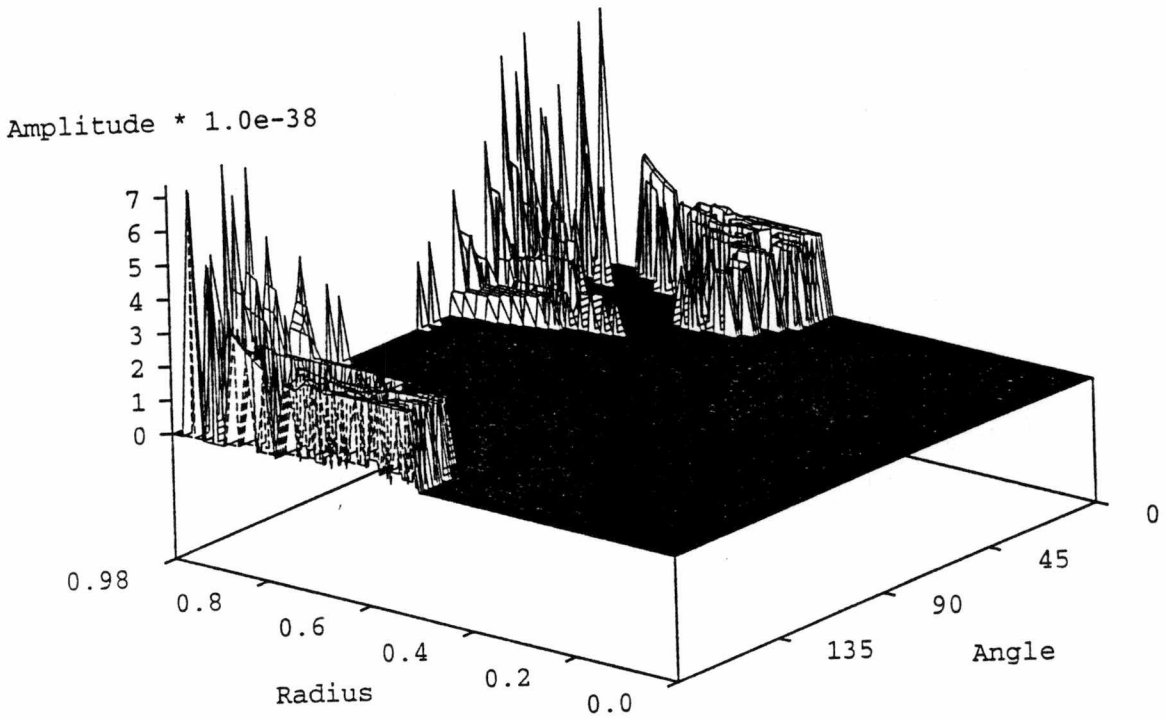

Figure (4.1) - Direct Form after 1 Million Samples (from r = 0.0)

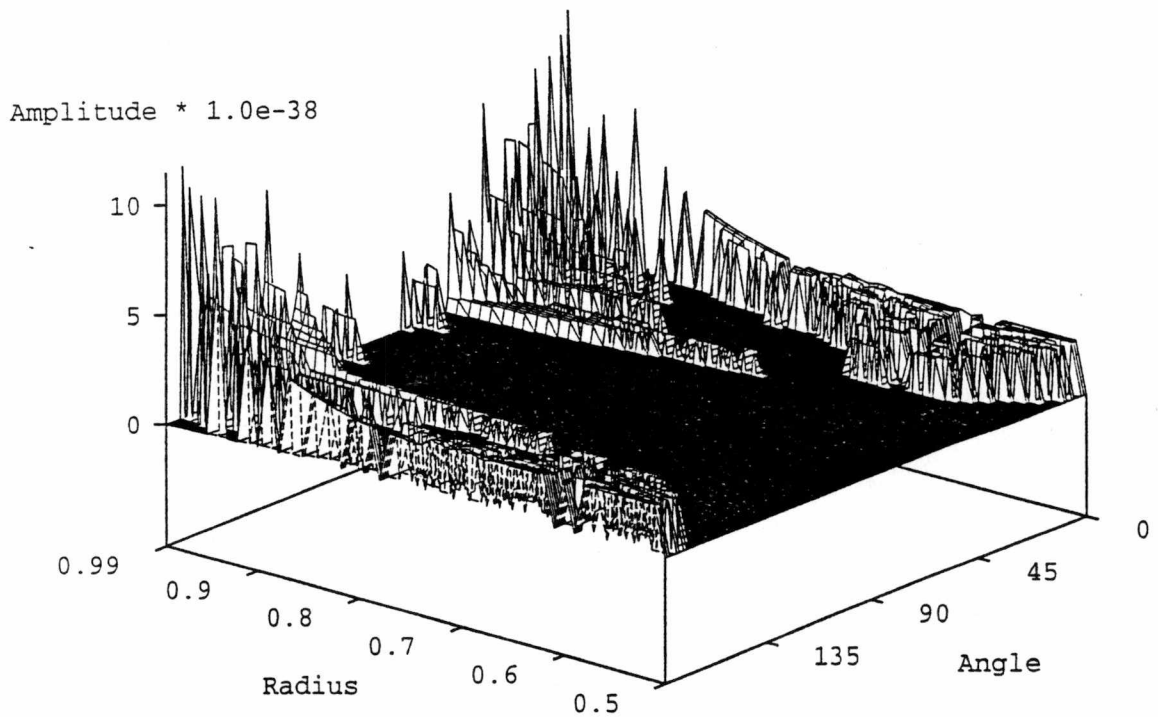

Figure (4.2) - Direct Form after 1 Million Samples (from r = 0.5)

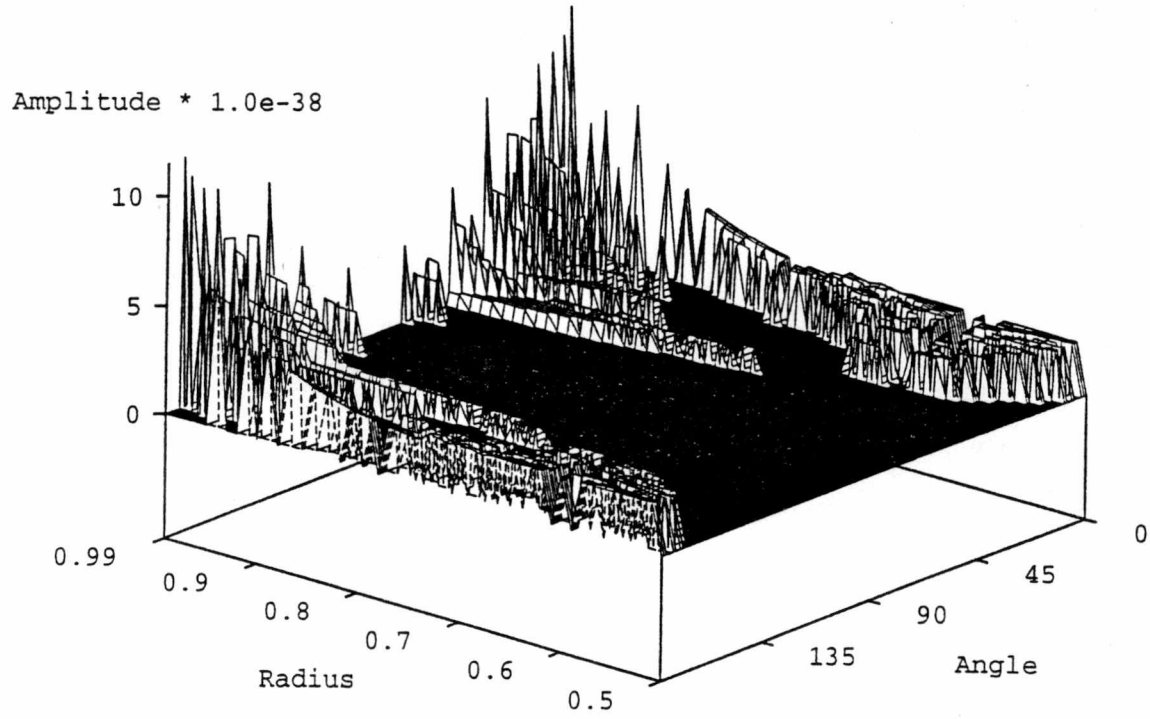

Figure (4.3) - Direct Form after 10 Million Samples (from r = 0.5)

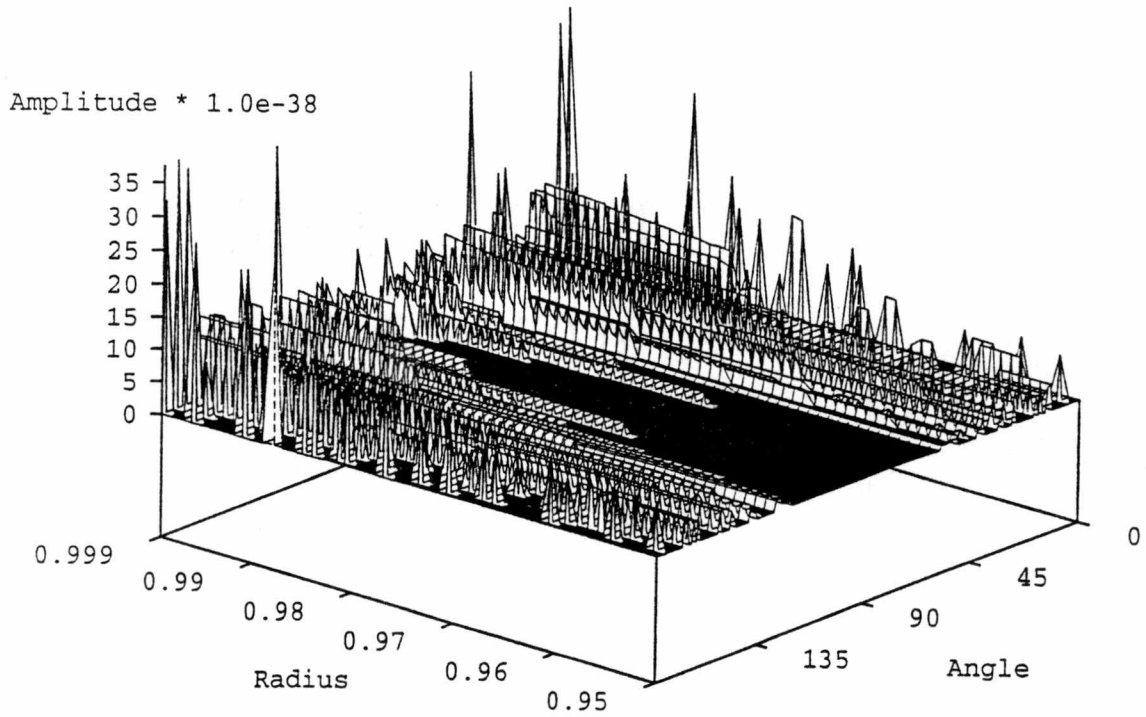

Figure (4.4) - Direct Form after 1 Million Samples (from r = 0.95)

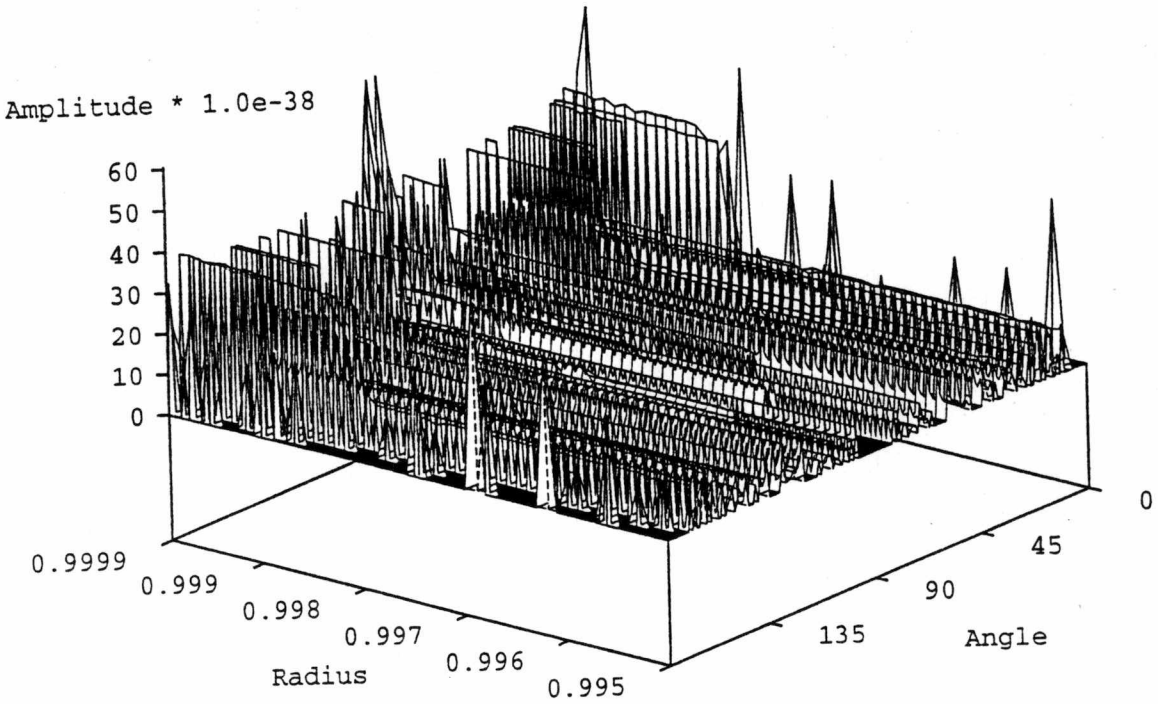

Figure (4.5) - Direct Form after 5 Million Samples (from r = 0.995)

From Figure (4.1), the Direct Form structure appears to have no limit cycles for radii less than 0.5. For radii out to around 0.9, limit cycle oscillations were observed at small angles only, as illustrated in Figure (4.2). Figure (4.3) demonstrates that a limit cycle characteristic has been found, because if this were not so, it would differ in appearance from Figure (4.2). With radii out to 0.999, Figure (4.4) shows that no limit cycle oscillations appear for angles near the imaginary axis (90°). Finally, Figure (4.5) illustrates that for the Direct Form with pole radii out to r=0.9999, limit cycles do exist for most pole angles, and the limit cycle amplitudes are less than around $60 \cdot 10^{-38}$.

Since the Minimum-Noise and Type III realizations produced virtually identical results, the following Figures (4.6) through (4.9) are the results of the limit-cycle search for both structures.

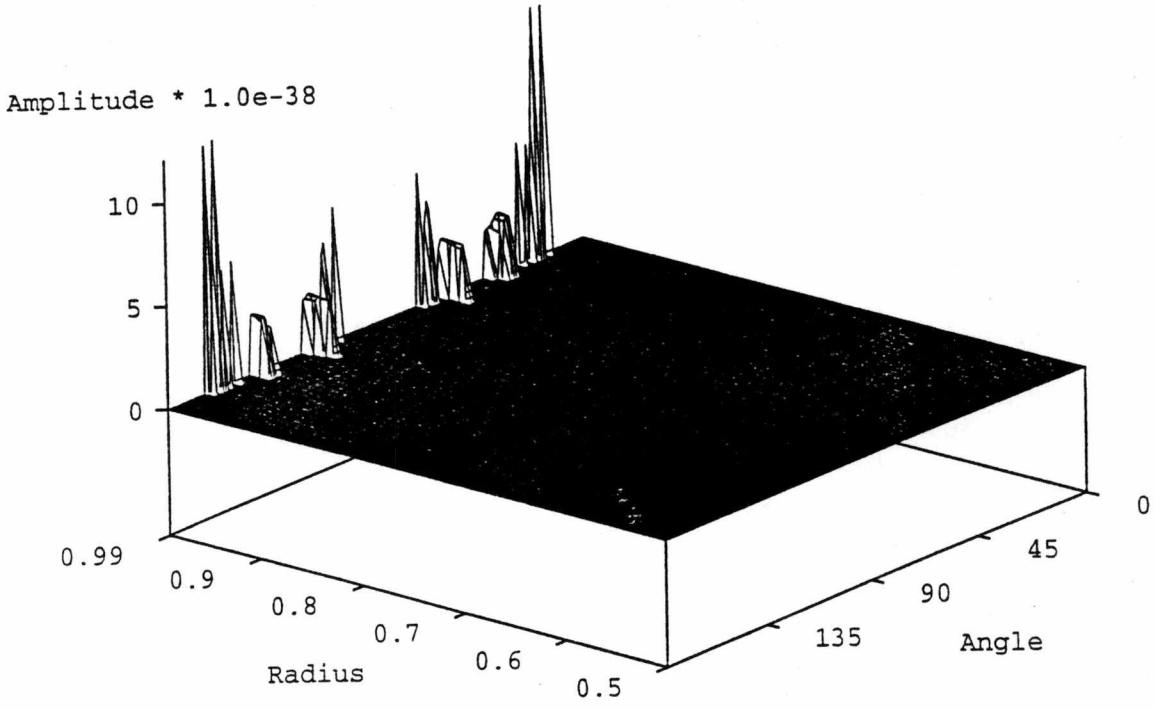

Figure (4.6) - Min.-Noise/Type III after 1 Million Samples (from r = 0.5)

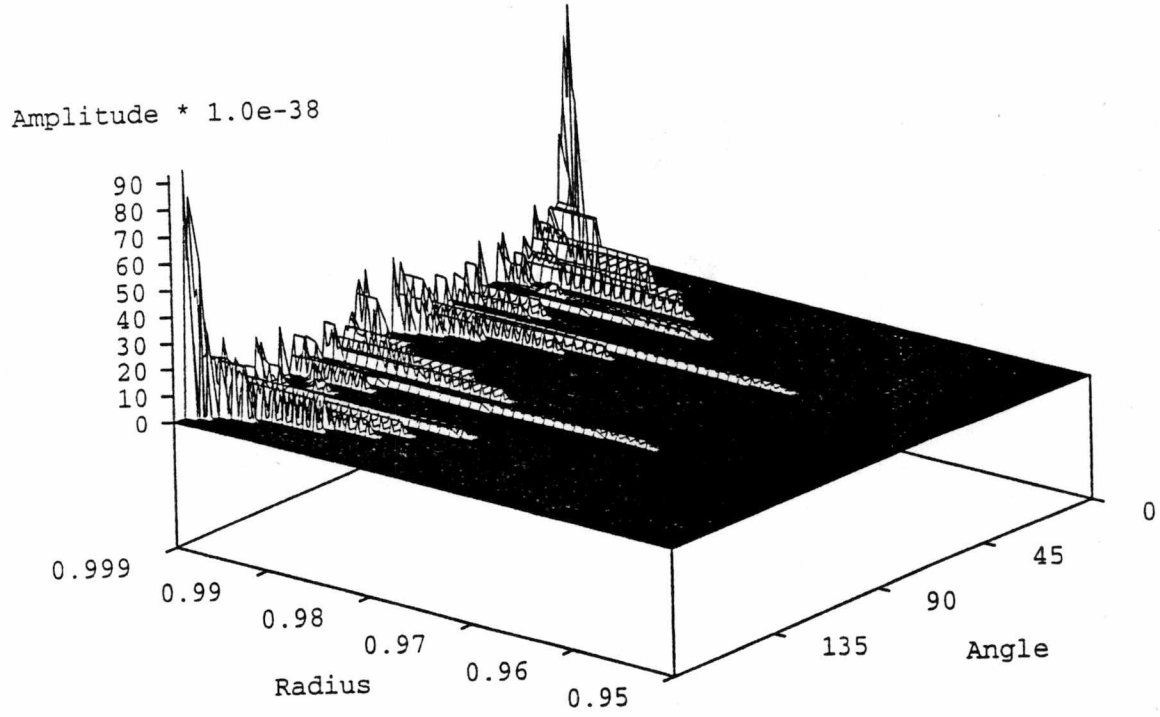

Figure (4.7) - Min.-Noise/Type III after 1 Million Samples (from r = 0.95)

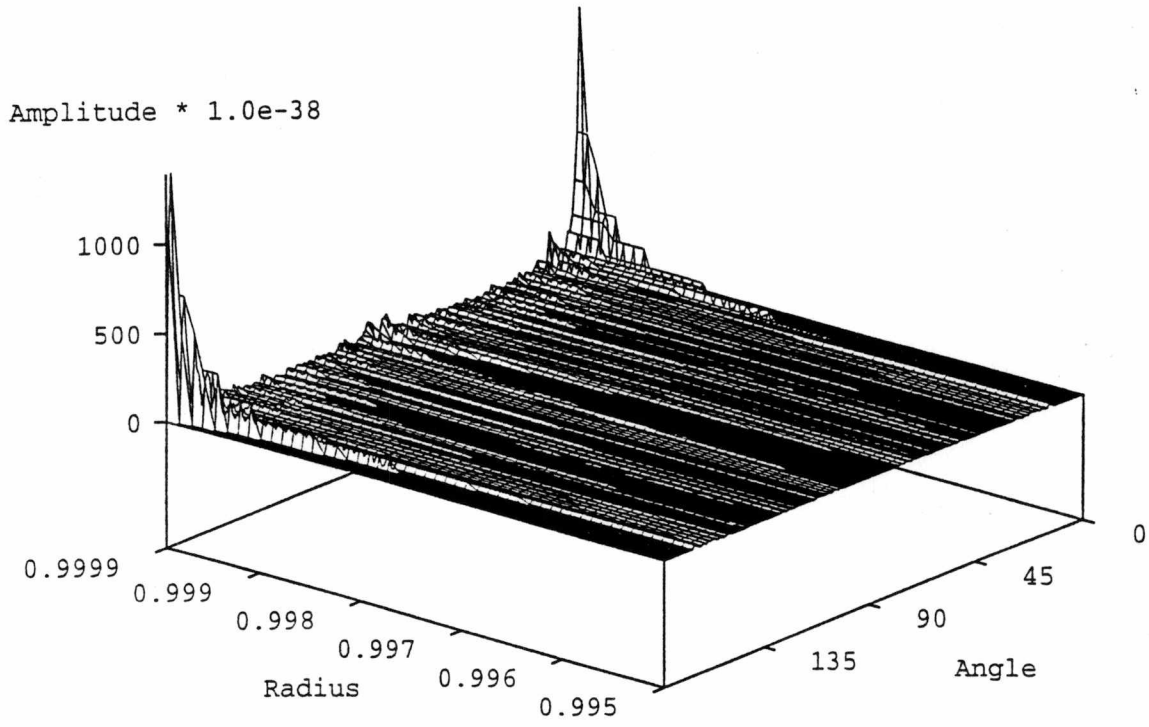

Figure (4.8) - Min.-Noise/Type III after 5 Million Samples (from r = 0.995)

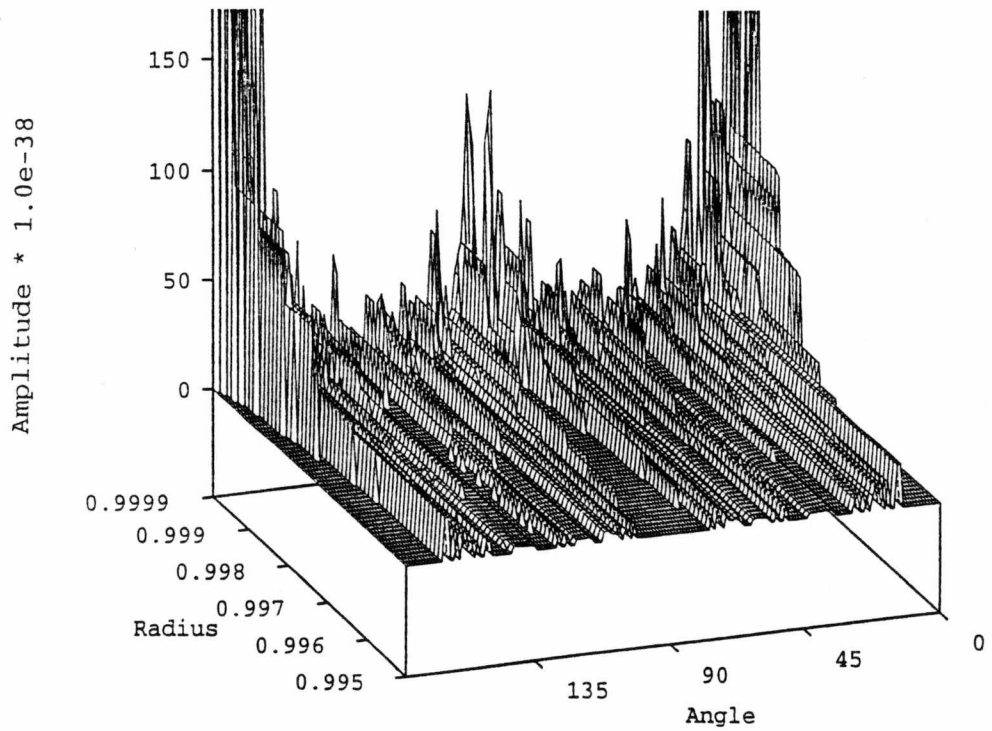

Figure (4.9) - Min.-Noise/Type III after 5 Million Samples (scaled, from r = 0.995)

Figures (4.6) and (4.7) show that no limit cycle oscillations were observed for Minimum-Noise and Type III filters if the pole radius was less than 0.965. The oscillations that occurred did so for angles of 61° and 119° only until the radius was increased to 0.98. Figures (4.8) and (4.9) show that limit cycle oscillations remain at low levels everywhere except where pole angles are small and pole radii are very close to 1.0. Limit-cycle amplitudes measured around $1400 \cdot 10^{-38}$ maximum for radii close to 1.0 and angles close to the real axis for both the Minimum-Noise and Type III structures.

A Closer Look at the Limit Cycles

In an attempt to explain why the three quantization methods produced virtually identical results, some additional experimental data was taken using the Type III realization with magnitude truncation and r=0.999, θ=165°. Since the Type III realization has $a_{11}=a_{22}$, based on (2.34), it should not have any limit cycles while using magnitude truncation quantization. This is because the generalized norm must decrease at each state update if $d_1^2=1$ and $d_2^2=|a_{12}/a_{21}|$ are used in generating the norm. However, limit cycles were observed for the Type III realization with all of the above conditions met. Therefore, either intermediate extended-precision quantizations or the underflow flush-to-zero characteristic of the processor must be responsible.

To determine which factor was responsible, the filter was applied for 1 million iterations, and then a starting point was recorded along with the result of every arithmetic operation for the next 1000 iterations. For comparison purposes, the same filter was applied from the recorded starting point on a PC with double-precision arithmetic. Since the PC can represent numbers that are vastly smaller than the DSP, it was assumed that the PC produced results that are of "infinite precision" compared to the DSP. The square of the generalized state vector length (as presented in Chapter II) was calculated for both filters by the PC at each filtering recursion and is illustrated in Figure (4.10).

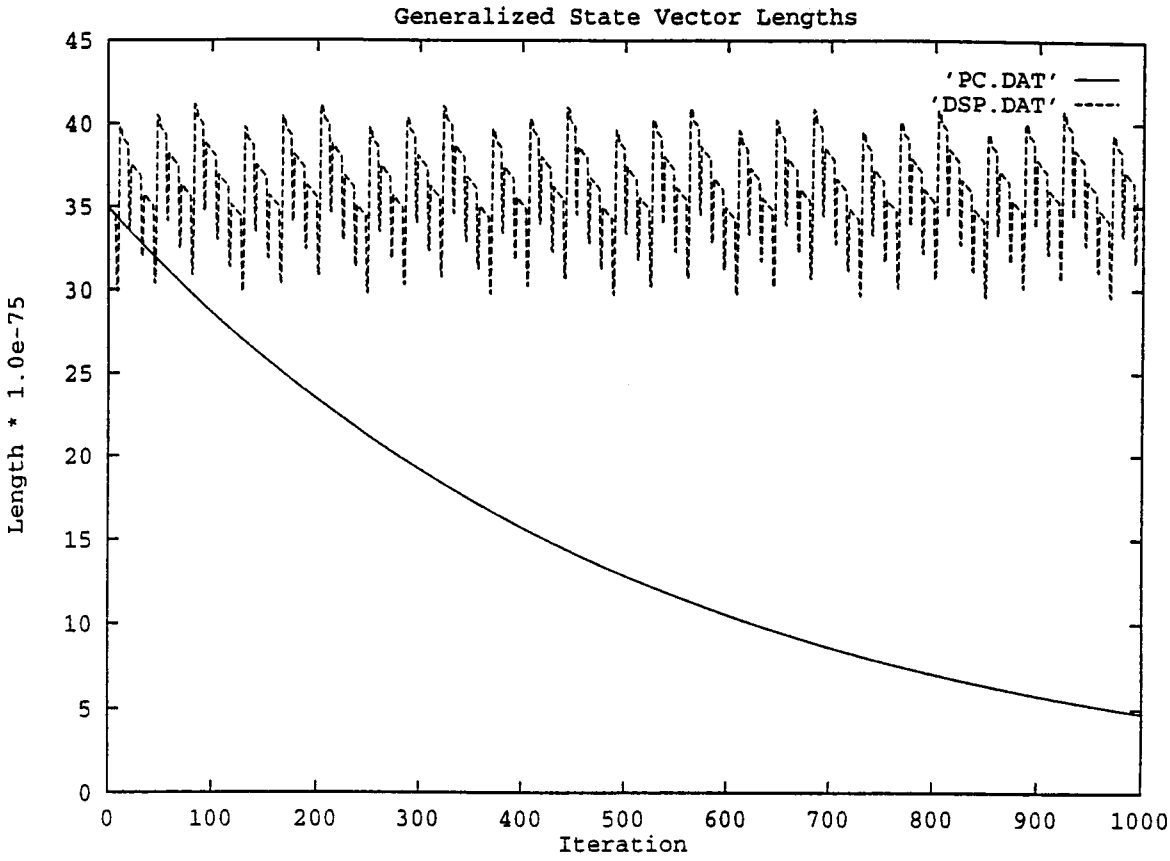

Figure (4.10) - Generalized Norm Squared for PC and DSP Filters

Observe that while the PC's generalized norm squared was monotonically decreasing through the entire sample interval as expected, the DSP's norm squared was oscillatory in nature. All of the conditions in (2.34), however, were met, so something else is causing the generalized norm to oscillate in the DSP. To get a better look, a close-up of the first 120 samples of Figure (4.10) is given in Figure (4.11).

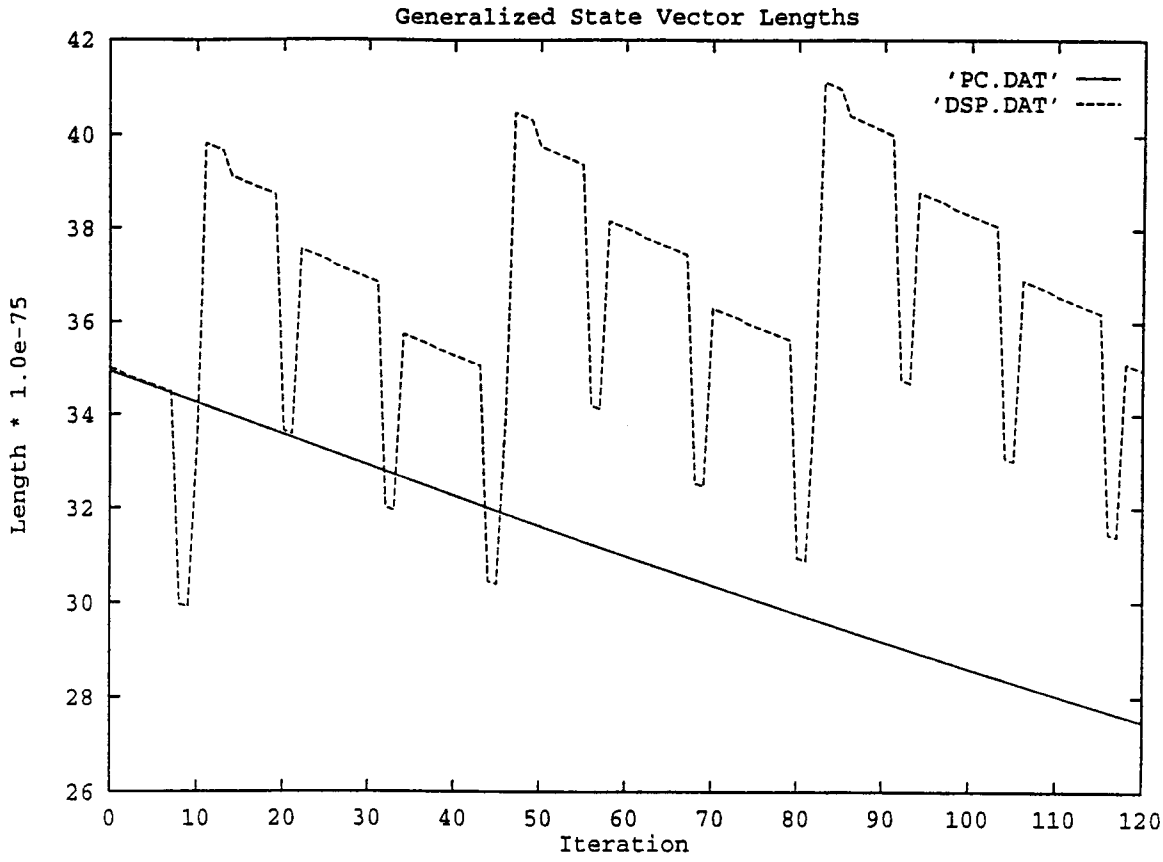

Figure (4.11) - Generalized Norm Squared for PC and DSP Filters (Close-up)

Notice that the squared generalized norms of the two filters are very close until around n=7, at which time the DSP's norm squared drops from 34.5e-75 to 30.0e-75 , and then later it jumps up to almost 40.0e-75. Figures (4.12) and (4.13) give listings of the results of all calculations which were performed by the DSP and by the PC in the first 49 samples illustrated in Figure (4.11). Note the places where the DSP flushed the result of an arithmetic operation to zero and how they relate to where the generalized norm squared experienced rapid changes.

Calculations Generated by DSP
Note: All values are multiplied by 1.0e+38 (except SVN Squared)

d2_squared	Filter Coefficients Z1	Z2
14.95809	-0.96496	-0.06685

n	s1(n)	s2(n)	X Z1*s1	s1(n+1) X+s2	Y Z2*s1	Z Z1*s2	s2(n+1) Y+Z	SVN Squared
0	15.89924	-2.55018	-15.3421	-17.8923	-1.06292	2.460825	1.397906	3.49E-74
1	-17.8923	1.397906	17.26536	18.66327	1.196163	-1.34892	0	3.48E-74
2	18.66327	0	-18.0093	-18.0093	-1.2477	0	-1.2477	3.48E-74
3	-18.0093	-1.2477	17.37826	16.13055	1.203984	1.203984	2.407969	3.47E-74
4	16.13055	2.407969	-15.5653	-13.1574	-1.07838	-2.32359	-3.40198	3.46E-74
5	-13.1574	-3.40198	12.69633	9.294355	0.879616	3.282771	4.162387	3.46E-74
6	9.294355	4.162387	-8.96868	-4.80629	-0.62136	-4.01654	-4.6379	3.45E-74
7	-4.80629	-4.6379	4.63788	0	0	4.475384	4.475384	3E-74
8	0	4.475384	0	4.475384	0	-4.31857	-4.31857	2.99E-74
9	4.475384	-4.31857	-4.31857	-8.63713	0	4.167243	4.167243	3.34E-74
10	-8.63713	4.167243	8.334486	12.50173	0	-4.02122	-4.02122	3.98E-74
11	12.50173	-4.02122	-12.0637	-16.0849	-0.83578	3.880318	3.044534	3.97E-74
12	-16.0849	3.044534	15.52127	18.56581	1.075331	-2.93785	-1.86252	3.97E-74
13	18.56581	-1.86252	-17.9153	-19.7778	-1.24119	1.79726	0	3.91E-74
14	-19.7778	0	19.08476	19.08476	1.322213	0	1.322213	3.9E-74
15	19.08476	1.322213	-18.416	-17.0938	-1.27588	-1.27588	-2.55177	3.9E-74
16	-17.0938	-2.55177	16.49485	13.94308	1.142781	2.462351	3.605132	3.89E-74
17	13.94308	3.605132	-13.4545	-9.84938	-0.93214	-3.47881	-4.41095	3.88E-74
18	-9.84938	-4.41095	9.50426	5.093309	0.658465	4.256391	4.914857	3.87E-74
19	5.093309	4.914857	-4.91484	0	0	-4.74264	-4.74264	3.36E-74
20	0	-4.74264	0	-4.74264	0	4.576457	4.576457	3.36E-74
21	-4.74264	4.576457	4.576457	9.152913	0	-4.4161	-4.4161	3.75E-74
22	9.152913	-4.4161	-8.83219	-13.2483	-0.6119	4.261356	3.649453	3.75E-74
23	-13.2483	3.649453	12.78407	16.43352	0.885694	-3.52158	-2.63588	3.74E-74
24	16.43352	-2.63588	-15.8577	-18.4936	-1.09864	2.54352	1.444882	3.73E-74
25	-18.4936	1.444882	17.84556	19.29044	1.236359	-1.39425	0	3.72E-74
26	19.29044	0	-18.6145	-18.6145	-1.28963	0	-1.28963	3.71E-74
27	-18.6145	-1.28963	17.96224	16.67261	1.244444	1.244444	2.488887	3.71E-74
28	16.67261	2.488887	-16.0884	-13.5995	-1.11462	-2.40168	-3.5163	3.7E-74
29	-13.5995	-3.5163	13.12298	9.606687	0.909175	3.393087	4.302261	3.69E-74
30	9.606687	4.302261	-9.27007	-4.96781	-0.64224	-4.15151	-4.79375	3.68E-74
31	-4.96781	-4.79375	4.793733	0	0	4.625776	4.625776	3.2E-74
32	0	4.625776	0	4.625776	0	-4.46369	-4.46369	3.19E-74
33	4.625776	-4.46369	-4.46369	-8.92738	0	4.30728	4.30728	3.57E-74
34	-8.92738	4.30728	8.61456	12.92184	0.596826	-4.15635	-3.55953	3.56E-74
35	12.92184	-3.55953	-12.4691	-16.0286	-0.86387	3.4348	2.57093	3.56E-74
36	-16.0286	2.57093	15.46694	18.03787	1.071566	-2.48084	-1.40928	3.55E-74
37	18.03787	-1.40928	-17.4058	-18.8151	-1.20589	1.359897	0	3.54E-74
38	-18.8151	0	18.15582	18.15582	1.257855	0	1.257855	3.53E-74
39	18.15582	1.257855	-17.5196	-16.2618	-1.21378	-1.21378	-2.42756	3.53E-74
40	-16.2618	-2.42756	15.69197	13.26441	1.087156	2.342497	3.429653	3.52E-74
41	13.26441	3.429653	-12.7996	-9.36997	-0.88677	-3.30948	-4.19625	3.51E-74
42	-9.36997	-4.19625	9.041643	4.845394	0.626415	4.049213	4.675627	3.5E-74
43	4.845394	4.675627	-4.67561	0	0	-4.51179	-4.51179	3.04E-74
44	0	-4.51179	0	-4.51179	0	4.353699	4.353699	3.04E-74
45	-4.51179	4.353699	4.353699	8.707398	0	-4.20115	-4.20115	3.4E-74
46	8.707398	-4.20115	-8.40229	-12.6034	0	4.053937	4.053937	4.05E-74
47	-12.6034	4.053937	12.16181	16.21575	0.842583	-3.91189	-3.0693	4.04E-74
48	16.21575	-3.0693	-15.6475	-18.7168	-1.08408	2.961754	1.877675	4.03E-74

Figure (4.12) - DSP Trace Listing

Calculations Generated from PC
Note: All values are multiplied by 1.0e+38 (except SVN Squared)

d2_squared	Filter Coefficients Z1	Z2
14.95809	-0.96496	-0.06685

n	s1(n)	s2(n)	X Z1*s1	s1(n+1) X+s2	Y Z2*s1	Z Z1*s2	s2(n+1) Y+Z	SVN Squared
0	15.89924	-2.55018	-15.3421	-17.8923	-1.06292	2.460825	1.397906	3.49E-74
1	-17.8923	1.397906	17.26536	18.66327	1.196163	-1.34892	-0.15276	3.49E-74
2	18.66327	-0.15276	-18.0093	-18.1621	-1.2477	0.147407	-1.1003	3.48E-74
3	-18.1621	-1.1003	17.52567	16.42537	1.214197	1.061743	2.27594	3.47E-74
4	16.42537	2.27594	-15.8498	-13.5739	-1.09809	-2.19619	-3.29428	3.47E-74
5	-13.5739	-3.29428	13.09825	9.803969	0.907461	3.178852	4.086313	3.46E-74
6	9.803969	4.086313	-9.46044	-5.37412	-0.65543	-3.94313	-4.59856	3.45E-74
7	-5.37412	-4.59856	5.185815	0.587257	0.359279	4.437424	4.796702	3.45E-74
8	0.587257	4.796702	-0.56668	4.230023	-0.03926	-4.62863	-4.66789	3.44E-74
9	4.230023	-4.66789	-4.0818	-8.74969	-0.28279	4.504323	4.221531	3.43E-74
10	-8.74969	4.221531	8.443098	12.66463	0.584947	-4.07361	-3.48866	3.42E-74
11	12.66463	-3.48866	-12.2209	-15.7095	-0.84667	3.366418	2.519744	3.42E-74
12	-15.7095	2.519744	15.15906	17.6788	1.050236	-2.43145	-1.38122	3.41E-74
13	17.6788	-1.38122	-17.0593	-18.4406	-1.18189	1.332818	0.150929	3.4E-74
14	-18.4406	0.150929	17.79439	17.94532	1.232815	-0.14564	1.087175	3.4E-74
15	17.94532	1.087175	-17.3165	-16.2293	-1.19971	-1.04908	-2.24879	3.39E-74
16	-16.2293	-2.24879	15.66066	13.41188	1.084988	2.169989	3.254977	3.38E-74
17	13.41188	3.254977	-12.9419	-9.68695	-0.89663	-3.14092	-4.03755	3.38E-74
18	-9.68695	-4.03755	9.347514	5.309961	0.647606	3.896077	4.543683	3.37E-74
19	5.309961	4.543683	-5.1239	-0.58022	-0.35499	-4.38447	-4.73946	3.36E-74
20	-0.58022	-4.73946	0.559886	-4.17957	0.03879	4.57339	4.61218	3.36E-74
21	-4.17957	4.61218	4.033122	8.645302	0.279419	-4.45057	-4.17115	3.35E-74
22	8.645302	-4.17115	-8.34237	-12.5135	-0.57797	4.024992	3.447023	3.34E-74
23	-12.5135	3.447023	12.07504	15.52207	0.836572	-3.32624	-2.48967	3.34E-74
24	15.52207	-2.48967	-14.9782	-17.4678	-1.0377	2.402429	1.364725	3.33E-74
25	-17.4678	1.364725	16.85577	18.22049	1.167786	-1.31691	-0.14912	3.32E-74
26	18.22049	-0.14912	-17.582	-17.7312	-1.2181	0.143894	-1.07421	3.32E-74
27	-17.7312	-1.07421	17.10986	16.03565	1.18539	1.036569	2.221958	3.31E-74
28	16.03565	2.221958	-15.4738	-13.2518	-1.07204	-2.1441	-3.21614	3.3E-74
29	-13.2518	-3.21614	12.78746	9.571319	0.885929	3.103446	3.989375	3.3E-74
30	9.571319	3.989375	-9.23594	-5.24656	-0.63988	-3.84959	-4.48946	3.29E-74
31	-5.24656	-4.48946	5.062724	0.573261	0.350751	4.332152	4.682903	3.28E-74
32	0.573261	4.682903	-0.55317	4.129729	-0.03832	-4.51881	-4.55714	3.28E-74
33	4.129729	-4.55714	-3.98502	-8.54216	-0.27609	4.397456	4.121369	3.27E-74
34	-8.54216	4.121369	8.242843	12.36421	0.571073	-3.97696	-3.40588	3.26E-74
35	12.36421	-3.40588	-11.931	-15.3369	-0.82659	3.28654	2.45995	3.26E-74
36	-15.3369	2.45995	14.79945	17.2594	1.025322	-2.37375	-1.34843	3.25E-74
37	17.2594	-1.34843	-16.6546	-18.0031	-1.15385	1.301182	0.147332	3.24E-74
38	-18.0031	0.147332	17.37223	17.51956	1.203567	-0.14217	1.061398	3.24E-74
39	17.51956	1.061398	-16.9057	-15.8443	-1.17124	-1.02421	-2.19545	3.23E-74
40	-15.8443	-2.19545	15.28909	13.09364	1.059245	2.118521	3.177766	3.22E-74
41	13.09364	3.177766	-12.6348	-9.45707	-0.87536	-3.06642	-3.94177	3.22E-74
42	-9.45707	-3.94177	9.125697	5.183925	0.632238	3.803652	4.43589	3.21E-74
43	5.183925	4.43589	-5.00228	-0.56639	-0.34656	-4.28046	-4.62702	3.21E-74
44	-0.56639	-4.62702	0.546543	-4.08048	0.037865	4.464889	4.502754	3.2E-74
45	-4.08048	4.502754	3.937497	8.44025	0.272794	-4.34498	-4.07218	3.19E-74
46	8.44025	-4.07218	-8.1445	-12.2167	-0.56426	3.929493	3.365233	3.19E-74
47	-12.2167	3.365233	11.78861	15.15385	0.816728	-3.24732	-2.43059	3.18E-74
48	15.15385	-2.43059	-14.6229	-17.0534	-1.01309	2.345419	1.332332	3.17E-74

Figure (4.13) - PC Trace Listing

Notice from Figure (4.12) that the sharp jumps in the state vector norm (SVN) Squared column on the right correspond directly to places where the results of arithmetic operations are flushed to zero. It appears that when $s_1(n+1)$ is flushed to zero, the generalized norm squared decreases sharply, and the result of Z_2*s_1 also flushes to zero. During subsequent iterations, the result of Z_2*s_1 continues to flush to zero for two or three more iterations. The generalized norm squared jumps up during the last iteration where Z_2*s_1 is flushed to zero. Given these observations, it can be concluded that the underflow flush-to-zero characteristic of the processor, and not the intermediate quantization of state variables, dominates in causing limit cycles in these filters.

The limit cycle period was measured for the Direct Form structure with poles at $r=0.7$ and $\theta=45°$ in the manner described in Chapter III. The two arrays were compared and found to be exactly the same with 35,135 samples between identical floating-point values. Floating-point limit cycles do repeat exact values, but because of the increased precision with which numbers can be represented, exact values are not repeated nearly as often as their fixed-point counterparts.

The initial state vector was changed twice in order to determine if the initial value of the state vector had any influence on the limit cycle behavior of the filter, but no observable difference was found between these results and those based on the impulse response with initial conditions at zero.

Chapter V - Summary and Conclusions

An experimental measurement and analysis of the limit-cycle characteristics was performed for causal, stable, second-order IIR filters with complex-conjugate poles realized by using three different second-order state-space filter structures . As expected, limit cycle oscillations were a function of both the filter's transfer function and its realization. However, the limit-cycle amplitudes were virtually the same for each of the quantization methods used: rounding, truncation, or magnitude truncation. The results in Chapter IV suggest that the nonlinearity introduced into the filter by the underflow flush-to-zero characteristic of the C31 had a much stronger influence on the limit cycle behavior of the filter than the final quantization of the state variables.

Given the advantage of its computational efficiency, the Type III structure appears to be a better choice than the Minimum-Noise structure when limit cycle characteristics are considered, since its limit-cycle amplitudes are the same as those of the Minimum-Noise realization, yet it requires less computation to implement. They both, however, have limit cycles which are larger than those of the Direct Form for small angles and radii close to the unit circle, so it may be that the Direct Form is the best realization for minimizing limit cycles for these pole locations. Given that the largest limit cycles for the Type III and Minimum-Noise structures were for poles which were very close to the real axis, it may be that making the poles real is the best solution of all, if the application permits.

Regardless of which filter realization is used, limit cycles in floating-point DSPs are largely due to the properties of the underflow flush-to-zero characteristic of the DSP and not necessarily due to the state variable quantization method used. If the underflow flush-to-zero characteristic were not used or available, and the filter were realized with

unnormalized floating-point numbers instead, the results would most likely be very different from those presented in this thesis. In many commercially available DSPs (including the C31), however, the underflow flush-to-zero characteristic cannot be disabled.

Avoiding limit cycles is difficult when the effects of underflow flush-to-zero are considered. It appears that floating-point implementations of low roundoff noise filters like the Minimum-Noise and Type III structures are not susceptible to limit-cycle oscillations when pole radii are less than 0.965. If the application does not require pole radii greater than 0.965 and one of these structures is being used to realize the filter, limit cycles can be disregarded completely. Otherwise, attention is only warranted if oscillations of $1400 \cdot 10^{-38}$ or less are significant and transfer function poles are very close to the unit circle (around 0.9999).

References

References

1. B.D. Green and L.E. Turner, "New Limit Cycle Bounds for Digital Filters," IEEE Transactions on Circuits and Systems, vol. 35, no. 4, pp. 365-374, April, 1988.

2. H.J. Butterweck, A.C.P. Van Meer, and G. Verkroost, "New Second-Order Digital Filter Sections without Limit Cycles," IEEE Transactions on Circuits and Systems, vol. CAS-31, no. 2, pp. 141-146, February, 1984.

3. S.R. Parker, "Limit-Cycle Oscillations in Digital Filters," IEEE Transactions on Circuit Theory, vol. CT-18, no. 6, pp. 687-697, November, 1971.

4. P.S.R. Diniz and A. Antoniou, "On the Elimination of Constant-Input Limit Cycles in Digital Filters," IEEE Transactions on Circuits and Systems, vol. CAS-31, no. 7, pp. 670-671, July, 1984.

5. B. Liu, "Effect of Finite Word Length on the Accuracy of Digital Filters - A Review," IEEE Transactions on Circuit Theory, vol. CT-18, no. 6, pp. 670-677, November, 1971.

6. K.T. Erickson, "Stability Analysis of Fixed-Point Digital Filters Using Computer Generated Lyapunov Functions -- Part I: Direct Form and Coupled Form Filters," IEEE Transactions on Circuits and Systems, vol. CAS-32, no. 2, pp. 113-132, February, 1985.

7. P.H. Bauer and J. Wang, "Limit Cycle Bounds for Floating Point Implementations of Second-Order Recursive Digital Filters," IEEE Transactions on Circuits and Systems II, vol. 40, no. 8, pp. 493-501, August 1993.

8. T. Kaneko, "Limit-Cycle Oscillations in Floating-Point Digital Filters," IEEE Transactions on Audio and Electroacoustics, vol. AU-21, no. 2, pp. 100-106, April, 1973.

9. B.W. Bomar, "New Second-Order State-Space Structures for Realizing Low Roundoff Noise Digital Filters," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-33, no. 1, pp. 106-110, February 1985.

10. L.B. Jackson, A.G. Lindgren, and Y. Kim, "Optimal Synthesis of Second-Order State-Space Structures for Digital Filters," IEEE Transactions on Circuits and Systems, vol. CAS-26, pp. 149-153, March 1979.

11. TMS320C3x User's Guide, Texas Instruments Incorporated, 1992.

12. W.T. Mills, C.T. Mullis, and R.A. Roberts, "Digital Filter Realizations Without Overflow Oscillations," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-26, pp. 334-338, August 1978.

13. J.G. Proakis and D.G. Manolakis, Digital Signal Processing Principles, Algorithms, and Applications, pp. 485-486, copyright 1992, Macmillan Publishing Company, New York, NY.

Appendix

Appendix A - Code

Direct Form Assembly Language Routine

```
*************************************************************************
;
;       FUNCTION: float direct_iir(float *coeffs, int iterations,
;                                  float *start, float *end, float input);
;
;       DESCRIPTION:  This c-callable function is written for the TMS320C3x
;                     DSP in order to process floating point filters.  This
;                     function will realize the Direct Form for a constant
;                     input (including zero input).
;
;       RETURNS: the output y[n] in R0.
;
;       ARGUMENTS:
;                     coeffs = an array of coefficients (2)
;                          z1=-a1, z2=-a2
;                     iterations = number of iterations to perform
;                     end = allocated memory to store final s[n] into
;                     start = the starting state vector
;                     input = the constant input
;
;       This function will realize the transfer function:
;                              -1
;                             z
;            H(z) = ----------------------
;                              -1       -2
;                     1 + a1*z   + a2*z
;
;       or equivalently,
;                      _t          -1  _
;            H(z) = c  * [zI-A]    * b
;
;       where A = [ z1=-a1   z2=-a2 ]     b = [ 1 ] * x[n]
;                 [    1        0    ]         [ 0 ]
;
;          _
;          c = [ 1 ]
;              [ 0 ]
;                                          _
```

```
;               This is realized by noting that:      { s[n] = [ s1[n] ]
;                                                                [ s2[n] ] }
;                                _    _    _
;                               s[n+1] = A * s[n] + b * x[n]
;                                      _t  _
;                               y[n] = c * s[n]
;
;               As a result, this program repeatedly steps through the algorithm:
;
;               NEXT STATE:
;                 s1[n+1] = z1*s1[n] + z2*s2[n] + x[n]
;                 s2[n+1] = s1[n]
;               OUTPUT:
;                 y[n] = s1[n]
;
;               After repeating the above 3 calculations *iterations* times, the
;               final value of y[n]=s1[n] is stored in *end*, and the value of
;               the output y[n] is passed back to the C program for analysis.
;
;               Author:       Dave Helsley
;               Revision:     2.00
;               Date:         December, 1994
;
*******************************************************************************
                .globl  _direct_iir

******* Set Up Mnemonics ***************************************************
ROUND   .set    0
y       .set    R0
s1      .set    R1
s2      .set    R2
z1      .set    R3
z2      .set    R4
input   .set    R5
coeffs  .set    AR1
FP      .set    AR3

                .text
_direct_iir:
                PUSH    FP              ; Save old frame pointer on stack
                LDI     SP,FP           ; Set new frame pointer

                PUSHF   R6              ; Save Registers
                PUSHF   R7              ;
```

```
******* Initialize ****************************************************

*** Starting State Variables ***
            LDI     *-FP(4),AR0         ; Temporary pointer
            LDF     *AR0,s1             ; Initialize s1[0]=start[0]
            LDF     *+AR0(1),s2         ; Initialize s2[0]=start[1]

*** Coefficient Pointer ***
            LDI     *-FP(2),coeffs      ; Load coefficient pointer
            LDF     *coeffs,z1
            LDF     *+coeffs(1),z2      ; Load coefficients

*** Input ***
            LDF     *-FP(6),input       ; Load input
            LDF     0.0, R6             ; Check for zero-input case
            CMPF    R6, input           ;
            BZD     ZERO                ; Delayed branch to zero-input algorthm

            LDI     *-FP(3),RC          ; Load counter
            SUBI    1,RC                ;  with # of iterations to do - 1.
            LDI     *-FP(5),AR0         ; Temporary pointer -> end[]

NONZERO:
            RPTB    ENDL                ;
******* Calculate NEXT STATE s[n+1] ************************************
;             s1[n+1] = z1*s1[n] + z2*s2[n] + x[n]
;             s2[n+1] = s1[n]
            MPYF    z1,s1,R7            ; z1 * s1[n] -> R7
            MPYF    z2,s2,R6            ; z2 * s2[n] -> R6
            ADDF    R7,R6               ; Add the two -> R6.
            ADDF    input,R6            ; Add in the input (R6=s1[n+1])

            .if   ROUND == 1            ; Conditional Assembly
            RND     s1,s2               ; Update s2[n+1] ** ROUNDING **
ENDL                                    ;                  ** ROUNDING **
            RND     R6,s1               ; Update s1[n+1] ** ROUNDING **

            .else
            STF     R6,*AR0             ; new s1 -> end[0] ** TRUNCATION **
            STF     s1,*+AR0            ; new s2 -> end[1] ** TRUNCATION **
            LDF     *AR0, s1            ; Load in new s1   ** TRUNCATION **
ENDL                                    ;                    ** TRUNCATION **
            LDF     *+AR0,s2            ; Load in new s2   ** TRUNCATION **
```

```
            .endif
            BR      TIDY_UP

ZERO:
            RPTB    ENDL2
******* Calculate NEXT STATE s[n+1] ************************************
;             s1[n+1] = z1*s1[n] + z2*s2[n] + x[n]
;             s2[n+1] = s1[n]
            MPYF    z1,s1,R7        ; z1 * s1[n] -> R7
            MPYF    z2,s2,R6        ; z2 * s2[n] -> R6
            ADDF    R7,R6           ; Add the two -> R6.

            .if  ROUND == 1         ; Conditional Assembly
            RND     s1,s2           ; Update s2[n+1] ** ROUNDING **
ENDL2                               ;                ** ROUNDING **
            RND     R6,s1           ; Update s1[n+1] ** ROUNDING **

            .else
            STF     R6,*AR0         ; new s1 -> end[0] ** TRUNCATION **
            STF     s1,*+AR0        ; new s2 -> end[1] ** TRUNCATION **
            LDF     *AR0, s1        ; Load in new s1   ** TRUNCATION **
ENDL2                               ;                  ** TRUNCATION **
            LDF     *+AR0,s2        ; Load in new s2   ** TRUNCATION **

            .endif
TIDY_UP:
******* Update the output state vector **********************************
            STF     s1,*AR0         ; s1 -> end[0]
            STF     s2,*+AR0        ; s2 -> end[1]

******* Calculate output y[n] ***************************************
;             y[n] = s1[n]
            RND     s1,y            ; Round to single-precision

            POPF    R7              ; Restore Registers
            POPF    R6              ;
            POP     FP              ; Restore Frame Pointer

            RETS                    ; Return from Subroutine
```

Minimum-Noise Assembly Language Routine

```
*************************************************************************
;
;       FUNCTION: float state_iir(float *coeffs, int iterations,
;                                 float *start, float *end, float input);
;
;       DESCRIPTION:  This c-callable function is written for the TMS320C3x
;                     DSP in order to process floating point filters.  This
;                     function will realize the State-Space Form for
;                     a constant input (including zero input).
;
;       RETURNS: TWICE the output y[n] in R0.
;
;       ARGUMENTS:
;                     coeffs = an array of coefficients (3)
;                          a11 and a22 (same), a12, a21
;                     iterations = number of iterations to perform
;                     end = allocated memory to store final s[n] into
;                     start = the starting state vector
;                     input = the constant input
;
;          A = [ r*cos(T)                        r*(cos(T)+1)   ]
;              [ -r*square(sin(T))/(cos(T)+1)    r*cos(T)       ]
;
;          _          _
;          c = [ 1 ] = b
;              [ 1 ]
;
;                                                  _
;       This is realized by noting that:       { s[n] = [ s1[n] ]
;                                                       [ s2[n] ] }
;                     _          _    _
;                     s[n+1] = A * s[n] + b * x[n]
;                           _t   _
;                     y[n] = c * s[n]
;
;       As a result, this program repeatedly steps through the algorithm:
;
;       NEXT STATE:
;         s1[n+1] = a11*s1[n] + a12*s2[n] + x[n]
;         s2[n+1] = a21*s1[n] + a22*s2[n] + x[n]
;       OUTPUT:
;         2*y[n] = s1[n] + s2[n]
;
```

```
;           After repeating the above 3 calculations *iterations* times, the
;           final value of y[n] is stored in *end*, and the value of
;           the output y[n] is passed back to the C program for analysis.
;
;           Author:     Dave Helsley
;           Revision:    2.00
;           Date:       November, 1994
;
*************************************************************************
            .globl  _state_iir

******* Set Up Mnemonics ************************************************
ROUND       .set        1
y           .set        R0
s1          .set        R1
s2          .set        R2
a11         .set        R3
a22         .set        R3
a12         .set        R4
a21         .set        R5
end         .set        AR5
FP          .set        AR3

            .text
_state_iir:
            PUSH        FP              ; Save old frame pointer on stack
            LDI         SP,FP           ; Set new frame pointer

            PUSH        R4              ;
            PUSH        R5              ;
            PUSHF       R6              ; Save Registers
            PUSHF       R7              ;
            PUSH        AR4             ;
            PUSH        AR5             ;
            PUSH        AR7             ;

******* Initialize ******************************************************

*** Starting State Variables ***
            LDI         *-FP(4),AR7     ; Temporary pointer
            LDF         *AR7,s1         ; Initialize s1[0]=start[0]
            LDF         *+AR7(1),s2     ; Initialize s2[0]=start[1]

*** Coefficient Pointer ***
```

```
        LDI     *-FP(2),AR0     ; *coeffs = {a11, a12, a21}
        LDF     *AR0++,a11      ; Load a11, a22
        LDF     *AR0++,a12      ; Load a12
        LDF     *AR0,a21        ; Load a21

*** Input ***
        LDF     *-FP(6),R6      ; Load input
        LDF     0,R0            ;
        CMPF    R0,R6           ; Check to see if input=0.0
        BZD     ZERO            ; Dealing with zero input -> use simpler
                                ;  realization

        LDI     *-FP(3),RC      ; Load counter
        SUBI    1,RC            ;  with # of iterations to do - 1.
        LDI     *-FP(5),end     ; Load end[] pointer

****************** SECTION FOR NON-ZERO INPUT ********************
NONZERO:
        RPTB    ENDL            ;
******* Calculate NEXT STATE s[n+1] ***********************************
;       NEXT STATE:
;       s1[n+1] = a11*s1[n] + a12*s2[n] + x[n]
;       s2[n+1] = a21*s1[n] + a22*s2[n] + x[n]
        MPYF3   a11,s1,R7               ; a11 * s1[n] -> R7
        MPYF3   a12,s2,R6               ; a12 * s2[n] -> R6
        ADDF3   R6,R7,R0                ; a11*s1[n]+a12*s2[n] -> R0

        MPYF3   a21,s1,R7               ; a21 * s1[n] -> R7
        MPYF3   a22,s2,R6               ; a22 * s2[n] -> R6
        ADDF3   R7,R6,s2                ;     a21*s1[n]+a22*s2[n]     ->
s2_AUX

        LDF     *-FP(6),R6              ; Load input

        ADDF3   R0,R6,s1                ; R0 + x[n] -> s1[n+1]
        ADDF    R6,s2                   ; s2_AUX + x[n] -> s2[n+1]

        .if  ROUND == 1
        RND     s1,s1                   ; Update s1 ** ROUNDING **
ENDL                                    ;             ** ROUNDING **
        RND     s2,s2                   ; Update s2 ** ROUNDING **

        .else
        STF     s1,*end                 ; ** TRUNCATION **
```

```
        STF     s2,*+end                ; ** TRUNCATION **
        LDF     *end,s1                 ; ** TRUNCATION **
ENDL                                    ; ** TRUNCATION **
        LDF     *+end,s2                ; ** TRUNCATION **

        .endif
        BR      TIDY_UP

****************** SECTION FOR ZERO INPUT ************************
ZERO:
        RPTB    ENDL2           ;
******* Calculate NEXT STATE s[n+1] ***********************************
;       NEXT STATE:
;         s1[n+1] = a11*s1[n] + a12*s2[n]
;         s2[n+1] = a21*s1[n] + a22*s2[n]
        MPYF3   a11,s1,R7       ; a11 * s1[n] -> R7
        MPYF3   a12,s2,R6       ; a12 * s2[n] -> R6
        ADDF3   R6,R7,R0        ; a11*s1[n]+a12*s2[n] -> R0

        MPYF3   a21,s1,R7       ; a21 * s1[n] -> R7
        MPYF3   a22,s2,R6       ; a22 * s2[n] -> R6
        ADDF3   R7,R6,s2        ; a21*s1[n]+a22*s2[n] -> s2

        .if     ROUND == 1
        RND     R0,s1           ; Update s1 ** ROUNDING **
ENDL2                           ;           ** ROUNDING **
        RND     s2,s2           ; Update s2 ** ROUNDING **

        .else
        STF     R0,*end         ; ** TRUNCATION **
        STF     s2,*+end        ; ** TRUNCATION **
        LDF     *end,s1         ; ** TRUNCATION **
ENDL2                           ; ** TRUNCATION **
        LDF     *+end,s2        ; ** TRUNCATION **

        .endif
TIDY_UP:
******* Update the output state vector ********************************
        STF     s1,*end                 ; s1 -> end[0]
        STF     s2,*+end                ; s2 -> end[1]
******* Calculate output y[n] *****************************************
;       OUTPUT:
;           2*y[n] = s1[n] + s2[n]
```

```
        ADDF3   s1,s2,y                 ; Place sum into y (R0)
        RND     y,y                     ; Round to single-precision

        POP     AR7                     ;
        POP     AR5
        POP     AR4                     ;
        POPF    R7                      ; Restore Registers
        POPF    R6                      ;
        POP     R5                      ;
        POP     R4                      ;
        POP     FP                      ; Restore Frame Pointer
        RETS                            ; Return from Subroutine
```

Type III Assembly Language Routine

```
*************************************************************************
;
;   FUNCTION: float type3_iir(float *coeffs, int iterations, float *start, float *end,
;                                        float input, float *trace);
;
;   DESCRIPTION:  This c-callable function is written for the TMS320C3x DSP in
;                 order to process floating point filters.  This function will realize
;                 the Type III State-Space Form for a constant input (including zero
;                 input).
;
;   RETURNS: the output y[n] in R0.
;
;   ARGUMENTS:
;       coeffs = an array of coefficients (2)
;            z1 = r*cos(theta), z2 = -(r*sin(theta))^2
;       iterations = number of iterations to perform
;       end = allocated memory to store final s[n] into
;       start = the starting state vector
;       input = the constant input
;
;       A = [ z1   1 ]      b = [ 0 ]
;           [ z2  z1 ]          [ 1 ]
;
;       c = [ z1 ]
;           [ 1  ]
;
;   This is realized by noting that:    { s[n] = [ s1[n] ]
;                                                [ s2[n] ] }
```

```
;          s[n+1] = A * s[n] + b * x[n]
;                   _t  _
;          y[n] = c * s[n]
;
;     As a result, this program repeatedly steps through the algorithm:
;
;     NEXT STATE:
;       s1[n+1] = z1*s1[n] + s2[n]
;       s2[n+1] = z2*s1[n] + z1*s2[n] + x[n]
;     OUTPUT:
;       y[n] = z1*s1[n] + s2[n]
;
;     After repeating the above 3 calculations *iterations* times, the
;     final value of y[n] is stored in *end*, and the value of
;     the output y[n] is passed back to the C program for analysis.
;
;     Author:       Dave Helsley
;     Revision:     3.00 Mag_Trunc
;     Date:         March, 1995
;
*************************************************************************
            .globl  _type3_iir

******* Set Up Mnemonics ************************************************
QUANT   .set    3       ; Quantization Method: 1=Round, 2=Truncate,
y       .set    R0      ;  3=Magnitude-Truncate, 0=No quantization
s1      .set    R1
s2      .set    R2
z1      .set    R3
z2      .set    R4
end     .set    AR5
FP      .set    AR3

            .data
MASK    .word   0FFFFFF00h

            .text
_type3_iir:
            PUSH    FP                  ; Save old frame pointer on stack
            LDI     SP,FP               ; Set new frame pointer

            PUSH    R4                  ;
            PUSH    R5                  ;
```

```
        PUSHF  R6               ; Save Registers
        PUSHF  R7               ;
        PUSH   AR4              ;
        PUSH   AR5              ;
        PUSH   AR7              ;

******* Initialize ***************************************************

*** Starting State Variables ***
        LDI    *-FP(4),AR7      ; Temporary pointer
        LDF    *AR7,s1          ; Initialize s1[0]=start[0]
        LDF    *+AR7(1),s2      ; Initialize s2[0]=start[1]

*** Coefficient Pointer ***
        LDI    *-FP(2),AR0      ; *coeffs = {z1, z2}
        LDF    *AR0,z1          ; Load z1
        LDF    *+AR0(1),z2      ; Load z2

*** Input ***
        LDF    *-FP(6),R5       ; Load input
        LDF    0,R0             ;
        CMPF   R0,R5            ; Check to see if input=0.0
        BZD    ZERO             ; Dealing with zero input -> use a simpler
                                ; realization

        LDI    *-FP(3),RC       ; Load counter
        SUBI   1,RC             ; with # of iterations to do - 1.
        LDI    *-FP(5),end      ; Load end[] pointer

****************** SECTION FOR NON-ZERO INPUT ********************
NONZERO:
        RPTB   ENDL             ;
******* Calculate NEXT STATE s[n+1] ***********************************
;   NEXT STATE:
;   s1[n+1] = z1*s1[n] + s2[n]
;   s2[n+1] = z2*s1[n] + z1*s2[n] + x[n]
        MPYF3  z1,s1,R7         ; z1 * s1[n] -> R7
        ADDF3  s2,R7,R0         ; z1*s1[n]+s2[n] -> R0

        MPYF3  z2,s1,R7         ; a21 * s1[n] -> R7
        MPYF3  z1,s2,R6         ; a22 * s2[n] -> R6
        ADDF   R7,R6            ; z2*s1[n]+z1*s2[n] -> R6
```

```
        ADDF3  R5,R6,s2              ; s2 + x[n] -> s2[n+1]

        .if    QUANT == 1
        RND    s2,s2                 ; ** ROUNDING **
ENDL                                 ; ** ROUNDING **
        RND    R0,s1                 ; ** ROUNDING **

        .elseif QUANT == 2
        AND    @MASK,s2              ; ** TRUNCATION **
ENDL                                  ; ** TRUNCATION **
        AND3   R0,@MASK,s1           ; ** TRUNCATION **

        .elseif QUANT == 3
        NEGF   s2,R6                 ; ** MAGNITUDE TRUNCATION **
        AND    @MASK,s2              ; X = Trunc(X) if (X > 0),
        AND    @MASK,R6              ;    -Trunc(-X) if (X <= 0).
        NEGF   R6,R6
        LDFN   R6,s2

        NEGF   R0,s1                 ; ** MAGNITUDE TRUNCATION **
        AND    @MASK,R0              ; X = Trunc(X) if (X > 0),
        AND    @MASK,s1              ;      -Trunc(-X) if (X <= 0).
        NEGF   s1,s1
ENDL
        LDFP   R0,s1

        .else
ENDL
        LDF    R0,s1                  ; ** NO QUANTIZATION **
        .endif
        BR     TIDY_UP

****************** SECTION FOR ZERO INPUT ********************
ZERO:
        RPTB   ENDL2          ;
******* Calculate NEXT STATE s[n+1] ************************************
;    NEXT STATE:
;    s1[n+1] = z1*s1[n] + s2[n]
;    s2[n+1] = z2*s1[n] + z1*s2[n]
        MPYF3  z1,s1,R7          ; z1 * s1[n] -> R7
        ADDF3  s2,R7,R0          ; z1*s1[n]+s2[n] -> R0

        MPYF3  z2,s1,R7          ; z2 * s1[n] -> R7
```

```
            MPYF3   z1,s2,R6          ; z1 * s2[n] -> R6
            ADDF    R7,R6             ; z2*s1[n]+z1*s2[n] -> R6

            .if     QUANT == 1
            RND     R6,s2             ; ** ROUNDING **
ENDL2                                 ; ** ROUNDING **
            RND     R0,s1             ; ** ROUNDING **

            .elseif QUANT == 2
            AND3    R6,@MASK,s2       ; ** TRUNCATION **
ENDL2                                 ; ** TRUNCATION **
            AND3    R0,@MASK,s1       ; ** TRUNCATION **

            .elseif QUANT == 3
            NEGF    R6,s2             ; ** MAGNITUDE TRUNCATION **
            AND     @MASK,R6          ; X = Trunc(X)   if (X > 0),
            AND     @MASK,s2          ;    -Trunc(-X) if (X <= 0).
            NEGF    s2,s2
            LDFP    R6,s2

            NEGF    R0,s1             ; ** MAGNITUDE TRUNCATION **
            AND     @MASK,R0          ; X = Trunc(X)   if (X > 0),
            AND     @MASK,s1          ;    -Trunc(-X) if (X <= 0).
            NEGF    s1,s1
ENDL2
            LDFP    R0,s1

            .else
            LDF     R6,s2             ; ** NO QUANTIZATION **
ENDL2
            LDF     R0,s1             ; ** NO QUANTIZATION **
            .endif

TIDY_UP:
******* Update the output state vector **********************************
            STF     s1,*end           ; s1 -> end[0]
            STF     s2,*+end          ; s2 -> end[1]

******* Calculate output y[n] ***************************************************
;       OUTPUT:
;       y[n] = z1*s1[n] + s2[n]  (actually, y[n+1])

            MPYF    z1,s1             ; z1*s1[n+1] -> s1
```

```
ADDF3   s1,s2,y          ; Place sum into y (R0)
RND     y,y              ; Round to single-precision

POP     AR7              ;
POP     AR5              ;
POP     AR4              ;
POPF    R7               ; Restore Registers
POPF    R6               ;
POP     R5               ;
POP     R4               ;

POP     FP               ; Restore Frame Pointer

RETS                     ; Return from Subroutine
```

Direct Form Limit Cycle Search C Routine

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

/* Function Prototypes */
#include "params.h"

#ifdef A
 #define THETA_1 1
 #define THETA_2 45
#endif
#ifdef B
 #define THETA_1 45
 #define THETA_2 90
#endif
#ifdef C
 #define THETA_1 90
 #define THETA_2 135
#endif
#ifdef D
 #define THETA_1 135
 #define THETA_2 180
#endif

#define  NUM_THETA  (THETA_2 - THETA_1)
#define  LENGTH     1000000                  /* Length of initial realization */
#define  NUM_R      50                       /* Number of Radius Points */
```

```
#define  R_START    0.5                              /* Starting Radius */
#define  R_INC      (((float) 1.0 - (float) R_START) / (float) NUM_R)

extern float direct_iir(float *coeffs, int iterations, float *start, float *end, float input);

float *largest_array;

void main ( void )
{       float output, dummy = (float) 0.0, SIN, COS;
        float *coeffs, *start, dc_out, r = (float) 0.0, theta = (float) 0.0;
        int i,j,k,n,l;                          /* Loop counters          */
        float pi_ov_180 = 0.017453292519943;    /* Angle increment of 1 degree  */
        int la_idx=-1;

        *TCTL1 = 6;
        *BUSCTL = 0;

        asm("   LDI 0,IF");                     /* Initialize Interrupt Flags */
        asm("   LDI 20h,IE");
        asm("   LDI 2800h,ST");

        coeffs = (float *) malloc(2*sizeof(float));
        start = (float *) malloc(2*sizeof(float));
        largest_array = (float *) malloc(NUM_R * NUM_THETA * sizeof(float));

        if( !coeffs || !start || !largest_array )
        {       BLINK();
                exit(1);
        }

        /* Initialize largest_array to floating-point zeros */
        for(i=0; i<NUM_R*NUM_THETA; i++)
                largest_array[i] = (float) 0.0;

        for(i=THETA_1; i<THETA_2; i++)
        {       theta = (float) i * pi_ov_180;
                SIN = sin(theta);
                COS = cos(theta);

                /* Signal Incrementing THETA */
                *TCTL1 = 2;
                for(k=0; k<500000; k++);
                *TCTL1 = 6;
```

```
            for(j=0; j<NUM_R; j++)
            {       r = (float) R_START + j * (float) R_INC;
                    la_idx++;

                    /* Signal Incrementing R */
                    *TCTL1 = 2;
                    for(k=0; k<50000; k++);
                    *TCTL1 = 6;

                    start[0] = (float) 0.0;
                    start[1] = (float) 0.0;
                    coeffs[0] = (float) 2.0 * r * COS;          /* z1=-a1 */
                    coeffs[1] = -r*r;                           /* z2=-a2 */

                    output = direct_iir(coeffs, 1, start, start, 1.0);
                    output = direct_iir(coeffs, LENGTH, start, start, 0.0);

                    for(n=0; n<1000; n++)
                    /* Find largest amplitude in next 1000 samples */
                    {       output = direct_iir(coeffs, 1, start, start, 0.0);
                            dummy = fabs( (float) output );
                            if( dummy > (float) largest_array[la_idx] )
                                    largest_array[la_idx] = dummy;
                    }
            }
        }
        BLINK();        /* Notify user that the search is complete */
        exit(0);
}
```

Minimum-Noise Limit Cycle Search C Routine

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

/* Function Prototypes */
#include "params.h"

#ifdef A
 #define THETA_1 1
 #define THETA_2 45
#endif
```

```
#ifdef B
 #define THETA_1 45
 #define THETA_2 90
#endif
#ifdef C
 #define THETA_1 90
 #define THETA_2 135
#endif
#ifdef D
 #define THETA_1 135
 #define THETA_2 180
#endif

#define  square(x)  ( (x) * (x) )
#define  LENGTH     5000000                    /* Length of initial realization */
#define  NUM_THETA  (THETA_2 - THETA_1) /* Number of angles to go through */
#define  NUM_R      50                         /* Number of radius points */
#define  R_START    0.995                      /* Starting Radius */
#define  R_INC      (((float) 1.0 - (float) R_START) / (float) NUM_R)

extern float state_iir(float *coeffs, int iterations, float *start, float *end, float input);

float *largest_array, r, theta, *coeffs;
int i, j;

void main ( void )
{       float output, *start, SIN, COS;
        int k,n;
        float pi_ov_180 = 0.017453292519943;  /* Angle increment of 1 degree  */
        int la_idx=-1;

        *TCTL1 = 6;
        *BUSCTL = 0;

        asm("   LDI 0,IF");                    /* Initialize Interrupt Flags */
        asm("   LDI 20h,IE");
        asm("   LDI 2800h,ST");

        coeffs = (float *) malloc(3*sizeof(float));
        start = (float *) malloc(2*sizeof(float));
        largest_array = (float *) malloc(NUM_R*NUM_THETA*sizeof(float));

        if( !coeffs || !start || !largest_array )
```

```
{       BLINK();
        exit(1);
}

/* Initialize largest_array to floating-point zeros */
for(i=0; i<NUM_R*NUM_THETA; i++)
        largest_array[i] = (float) 0.0;

for(i=THETA_1; i<THETA_2; i++)
{       theta = (float) i * pi_ov_180;
        SIN = sin(theta);
        COS = cos(theta);

        /* Signal Incrementing THETA */
        *TCTL1 = 2;
        for(k=0; k<500000; k++);
        *TCTL1 = 6;

        for(j=0; j<NUM_R; j++)
        {  r = (float) R_START + (float) j * (float) R_INC;
                la_idx++;

                /* Signal Incrementing R */
                *TCTL1 = 2;
                for(k=0; k<50000; k++);
                *TCTL1 = 6;

                start[0] = (float) 0.0;
                start[1] = (float) 0.0;

                coeffs[0] = (float) r * (float) COS;                    /* a11, a22 */
                coeffs[1] = (float) r + coeffs[0];                      /* a12 */
                coeffs[2] = -r * square(SIN) / (COS + (float) 1.0);     /* a21 */

                output = 0.5*state_iir(coeffs, 1, start, start, (float) 1.0);
                output = 0.5*state_iir(coeffs, LENGTH, start, start, (float) 0.0);

        /* Find largest amplitude in next 1000 samples */
                for(n=1; n<=1000; n++)
                {  output = 0.5*state_iir(coeffs, 1, start, start, (float) 0.0);
                        if( fabs( (float)output) > (float) largest_array[la_idx] )
                                largest_array[la_idx] = fabs( (float)output);
                }
```

```
            }
        }
        BLINK();
        exit(0);
}
```

Type III Limit Cycle Search C Routine

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

/* Function Prototypes */
#include "params.h"

#ifdef A
 #define THETA_1 1
 #define THETA_2 45
#endif
#ifdef B
 #define THETA_1 45
 #define THETA_2 90
#endif
#ifdef C
 #define THETA_1 90
 #define THETA_2 135
#endif
#ifdef D
 #define THETA_1 135
 #define THETA_2 180
#endif

#define  square(x)  ( (x) * (x) )
#define  LENGTH     5000000                    /* Length of initial realization */
#define  NUM_THETA  (THETA_2 - THETA_1) /* Number of angles to go through */
#define  NUM_R      50                         /* Number of radius points */
#define  R_START    0.995                      /* Starting radius */
#define  R_INC      (((float) 1.0 - (float) R_START) / (float) NUM_R)

extern float type3_iir(float *coeffs, int iterations, float *start, float *end, float input);

float *largest_array, r, theta, *coeffs;
int i, j;
```

```
void main ( void )
{       float output, *start, SIN, COS;
        int k,n;
        float pi_ov_180 = 0.017453292519943;       /* Angle increment of 1 degree */
        int la_idx=-1;

        *TCTL1 = 6;
        *BUSCTL = 0;

        asm("   LDI 0,IF");                        /* Initialize Interrupt Flags */
        asm("   LDI 20h,IE");
        asm("   LDI 2800h,ST");

        coeffs = (float *) malloc(2*sizeof(float));
        start = (float *) malloc(2*sizeof(float));
        largest_array = (float *) malloc(NUM_R*NUM_THETA*sizeof(float));

        if( !coeffs || !start || !largest_array )
        {       BLINK();
                exit(1);
        }

        /* Initialize largest_array to floating-point zeros */
        for(i=0; i<NUM_R*NUM_THETA; i++)
                largest_array[i] = (float) 0.0;

        for(i=THETA_1; i<THETA_2; i++)
        {       theta = (float) i * pi_ov_180;
                COS = cos(theta);
                SIN = sin(theta);

                /* Signal Incrementing THETA */
                *TCTL1 = 2;
                for(k=0; k<500000; k++);
                *TCTL1 = 6;

                for(j=0; j<NUM_R; j++)
                {       r = (float) R_START + (float) j * (float) R_INC;
                        la_idx++;

                        /* Signal Incrementing R */
                        *TCTL1 = 2;
                        for(k=0; k<50000; k++);
```

```
                    *TCTL1 = 6;

                    start[0] = 0.0;                    /* Load starting vector */
                    start[1] = 0.0;

                    coeffs[0] = (float) r * (float) COS;        /* z1 */
                    coeffs[1] = -square(r*SIN);                 /* z2 */

                    output = type3_iir(coeffs, 1, start, start, (float) 1.0);
                    output = type3_iir(coeffs, LENGTH, start, start, (float) 0.0);

                        /* Find largest amplitude in next 1000 samples */
                    for(n=0; n<1000; n++)
                    { output = type3_iir(coeffs, 1, start, start, (float) 0.0);
                        if( fabs( (float)output) > (float) largest_array[la_idx] )
                            largest_array[la_idx] = fabs( (float)output);
                    }
                }
            }
            BLINK();
            exit(0);
}
```

Vita

H. David Helsley, Jr. was born in Charleston, West Virginia on September 14, 1971. He received the Bachelor of Science degree in Electrical Engineering from Ohio Northern University, Ada, Ohio, in 1993, graduating with high distinction. He then worked under Dr. Bruce W. Bomar at The University of Tennessee Space Institute, Tullahoma, Tennessee for two academic years as a Graduate Research Assistant. During that time, he worked to support the CADDMAS (Computer-Aided Dynamic Data Monitoring and Acquisition System) Project sponsored by Sverdrup Technology, Incorporated at Arnold Engineering Development Center, Arnold Air Force Base, Tennessee.

This thesis is presented as part of the requirements for the Master's Degree program in Electrical Engineering at UTSI.

David Helsley is a member of Tau Beta Pi and Phi Eta Sigma.