Sub-nyquist sampling techniques

Bagshaw, Paul Christopher

SUB-NYQUIST SAMPLING TECHNIQUES

P C Bagshaw, B.Sc. (Hons)
School of Engineering and Applied Science
University of Durham, UK

Thesis for Master of Science submitted September 1990

ABSTRACT

A number of novel theoretical methods have been developed in an attempt to analyse data produced by sampling a signal at below the Nyquist rate and the limitations of the approaches have been investigated.

A technique is developed that allows, under specified conditions, the frequency and amplitude of a band-limited sinusoidal signal (with no harmonics) to be determined when the signal is sampled simultaneously with three uniform samplers at below the Nyquist rate. The three samplers operate at slightly different rates. Each has its output ideally low-pass filtered with a cut-off frequency at half the sampling rate. The frequencies of the signals output from the ideal filters are analysed to determine the input sinusoid parameters. The frequency of the sinusoid can also be found within a calculated tolerance when approximate filter output frequencies are known.

Two approaches extending this technique for a band-limited periodic signal consisting of more than just the fundamental, enable the frequencies of the harmonics to be found for the signal, but there is the possibility that other erroneous harmonics may be identified as part of the signal. The probability of this occurring can be reduced by uniformly sampling simultaneously with a greater number of samplers. This probability cannot reach zero. Furthermore, as the number of samplers increases or the number of signal harmonics increases, the computational workload imposed in determining the harmonic frequencies rises dramatically. The approaches are rendered impractical and sampling at irregular intervals is suggested as an alternative to using a very large number of uniform samplers.

A modified discrete Fourier transform and its inverse are developed to allow an estimated spectral analysis of a continuous periodic signal sampled at irregular intervals. Additive pseudo-random sampling and periodic sampling with dither are rigourously defined as two proposed irregular sampling schemes. The periodicity and symmetrical properties of the modified transform are derived for the two schemes. Consistently alias-free spectral analysis of a band-limited periodic signal is demonstrated using additive pseudo-random sampling with a maximum sampling rate below the Nyquist rate. This does not apply when using periodic sampling with dither.

SUB-NYQUIST SAMPLING TECHNIQUES

Paul Christopher Bagshaw, B.Sc. (Hons)

Master of Science Thesis

University of Durham, UK
School of Engineering and Applied Science

September 1990

CONTENTS

APPENDICES

DECLARATION

This thesis is submitted to the Board of Examiners for the School of Engineering and Applied Science at the University of Durham for the degree of Master of Science. The material contained within it is solely the original work of the author and when detailed reference has been made to other texts, the source of information has been clearly stipulated.

The author wishes to convey special thanks to Dr. Mansoor Sarhadi for his supervision and funding of the research undertaken, friends at the Queen's Head Hotel, Gilesgate, its 'spiritual' enlightenment and his parents and family for their moral support, without whom the production of this work might not have been possible.

GLOSSARY OF ABBREVIATIONS AND SYMBOLS

DFT	Discrete Fourier Transform.
DSPs	Digital Signal Processors.
DUX	Demultiplexing module.
FFT	Fast Fourier Transform.
LPF	Low-Pass Filter.
MTS	Michigan Terminal System.
MUSE	Multiple Sub-Nyquist sampling Encoding. Used in Japanese-developed high-definition television system.
NAG	Numerical Algorithms Group.
NTSC	National Television Systems Committee. US-developed colour television, using quadrature amplitude modulation of a colour sub-carrier and a luminance signal.
PAL	Phase Alternating Line. German-developed colour television system.
PC	Personal Computer (IBM compatible.)
PCM	Pulse Code Modulation. A development from pulse modulation involving sampling a continuous signal, quantizing the samples to specific levels, and encoding these values into some numerical form.
PRNG	Pseudo-Random Number Generator.
$\hat{a}_{fn}$	estimated real Fourier coefficient relating to the harmonic of frequency f_n.
A_m	amplitude of m^{th} signal harmonic.
a_m	real Fourier coefficient relating to m^{th} signal harmonic.
B_{err}	operational bandwidth for dealiasing algorithm when errors in f_o are considered.
$\hat{b}_{fn}$	estimated imaginary Fourier coefficient relating to the harmonic of frequency f_n.
b_m	imaginary Fourier coefficient relating to m^{th} signal harmonic.

B_{opt}	optimum operational bandwidth for dealiasing algorithm.
B_{pseudo}	bandwidth limitation required for input of pseudo-random DFT.
χ	pseudo-random variable for selection of τ.
$\delta(x)$	the impulse function.
δf	DFT frequency bin spacing.
dfo	frequency error in f_o.
dfs_1	difference in sampling frequencies f_{s1} and f_{s2}.
dfs_2	difference in sampling frequencies f_{s2} and f_{s3}.
dfs_3	difference in sampling frequencies f_{s1} and f_{s3}.
f_m	frequency of m^{th} signal harmonic.
f_n	frequency for which Fourier coefficients are calculated.
f_o	frequency of signal output by sampler/LPF system.
f_p	periodicity of $X_r(f)$.
FR_{12}, FR_{23}, FR_{13}	folding frequency/point of symmetry in aliasing pattern.
f_s	rate of uniform sampling.
f_x	frequency of a pure sinusoidal signal.
f_ϕ	fundamental frequency of a periodic function.
h(t)	rectangular window function.
L(f)	frequency response of an ideal low-pass filter.
M	number of components (fundamental plus harmonics) that make up a periodic signal.
N	number of consecutive samples taken of a signal.
P	number of possible values τ may take.
Q	minimum length of sequence $x(\lambda)$.
θ_m	relative phase of m^{th} signal harmonic.
R ()	independent pseudo-random number generating function. $0 < R() < 1$.
R_i	frequency resolution with which the input signal can be analysed.
R_u ()	pseudo-random number generating function with an uniform distribution.
τ	independent pseudo-random variable.

t_k	time of the $(k + 1)^{th}$ sampling instance
τ_{min}	minimum possible difference between one sampling instance and the next.
t_{N-1}	time final sample is taken.
T_s	uniform interval between regular sampling instances.
T_w	width of rectangular window function h(t).
T_ϕ	repetition interval of a periodic function.
$U_0(f)$	the continuous Fourier transform of $u_0(t)$.
$u_0(t)$	infinite series of impulses where each impulse corresponds to a sampling instance.
$U_1(f)$	the continuous Fourier transform of $u_1(t)$.
$u_1(t)$	repetition in the time domain corresponding to a series of impulses in the frequency domain.
W_1, W_2	relative workloads for algorithms $N^{o}\cdot 1$ & $N^{o}\cdot 2$ respectively.
W_{fft}	relative workload imposed by FFT.
ξ	number of values instance t_k can take.
$x(\lambda)$	uniform sequence accommodating every irregular sampling instance.
$x_a'(t)$	instantaneous sample values according to inverse pseudo-random DFT.
$X_a(f)$	the continuous Fourier transform of $x_a(t)$.
$x_a(t)$	function of time representing the magnitude of a continuous analogue signal.
$x_d(n)$	function representing the magnitude of a discrete analogue signal at the $(n+1)^{th}$ sample.
$X_o(f)$	the continuous Fourier transform of $x_o(t)$.
$x_o(t)$	function of time representing a continuous analogue signal output by a sampler/LPF system.
$X_r(f)$	estimated coefficients of signal $x_r(t)$ according to pseudo-random DFT.
$x_r(t)$	sampled, truncated, periodic waveform.

1. INTRODUCTION

Reliable and relatively inexpensive digital hardware has been used to perform signal processing tasks in preference to conventional analogue means. However, digital signal processing is not necessarily the best solution for all signal processing problems and often for extremely wide bandwidth signal real-time processing, analogue techniques are employed. The use of digital systems has many well known advantages over analogue systems, in particular in being able to provide a greater degree of flexibility in system design. It is therefore desirable to devise digital techniques which allow wide bandwidth signals to be rapidly processed in preference to analogue means.

Traditionally, the input to digital systems is formed by taking samples of a band-limited signal at a rate which is greater than or equal to twice the signal bandwidth; that is, at a Nyquist rate. If a signal is sampled at regular intervals below the Nyquist rate (at a sub-Nyquist rate) a phenomenon known as aliasing occurs.

The limited bandwidth of digital signal processors (currently approximately 25 MHz but forever increasing) prohibits the digital analysis of very high frequency signals, such as radar. Sampling such signals uniformly below the Nyquist rate inherently results in aliasing and a loss of information. A technique is required which will allow relatively slow digital signal processors (DSPs) to analyse wide bandwidth signals sampled below the Nyquist rate. The technique must therefore either resolve the ambiguities in the alias signal or somehow prevent the aliasing phenomenon.

This thesis describes the study of two methods aimed at resolving the problems of sub-Nyquist sampling. The first method involves the development of a technique which would ideally eliminate all the ambiguities in the aliased signal, and the second proposes a way to prevent the aliasing phenomenon. The origin of the aliases due to sampling a signal at uniform intervals at a sub-Nyquist rate is investigated in the frequency domain. A dealiasing algorithm is initially defined for a system of

sub-Nyquist samplers, to enable the elimination of ambiguities caused by sampling a pure sinusoidal signal (one consisting of its fundamental harmonic alone) with the system. An investigation is then made into extending the dealiasing algorithm to allow ambiguities caused by sub-Nyquist sampling of a signal containing any number of harmonics to be eliminated. The method used to avoid the aliasing phenomenon, rather than resolve it, centres around taking samples of a signal at irregularly spaced time intervals, as opposed to uniform sampling. Generators of irregularly spaced sampling instances are developed and a technique to perform consistently alias-free spectral analysis of the irregularly spaced samples is investigated. Simulation of every aspect of the methods studied is implemented to help in analysing their performance and efficiency, and aid in determining their limitations.

The techniques developed, if rigourous and truly capable of functioning at a sub-Nyquist rate, will inevitably have some limitations. Digital processing of analogue signals has its drawbacks. The conversion of an analogue signal to a digital form involves sampling the signal and quantizing the samples, resulting in distortion which inhibits the reconstruction of the original analogue signal from the quantized samples. Some detailed study of the limitations of the techniques is presented in this thesis.

Some work has already been done in the field of sub-Nyquist sampling. However, the techniques already well developed have been for specific applications; such as multiple sub-Nyquist sampling encoding (MUSE) for the Japanese high-definition television. The MUSE system covers a vast area of research and has not been included in this study of sub-Nyquist sampling techniques.

Another application specific technique has been devised for sub-Nyquist-encoded PCM NTSC colour television **[1, Rossi]** that enables the data rate of a PCM colour television signal to be reduced. The encoding frequency is reduced below the Nyquist rate such that the lower sidebands of the television signal overlap the baseband video frequencies. The sub-Nyquist encoding frequency is carefully chosen such that the alias components

are placed into parts of the spectrum not normally occupied by the luminance or chrominance components of the video signal. A proper choice of comb filters (having a multiplicity of regularly spaced narrow attenuation bands) then allow most of the alias signals to be removed from the baseband video.

Other systems make use of aliases caused by sub-Nyquist sampling rather than attempt to resolve them. A new despreading method based on sub-Nyquist sampling **[2, Führen & Den Dulk]** uses the aliasing phenomenon to make the input and output bands of the new despreader different, in accordance with the distinguished despreading method based on heterodyne correlation. This thesis does not consider such application dedicated techniques, but addresses the problem of eliminating all the aliasing effects caused by sampling a signal uniformly at a sub-Nyquist rate.

Previous research related to solving the problem of aliasing in systems using sub-Nyquist sampling, is limited. Some work is related to resolving the frequency ambiguities resulting from uniform sampling at a sub-Nyquist rate, while other works address random sampling schemes which could, in an ideal world, allow consistently alias-free spectral analysis of a non-band-limited signal. The relevant works are referenced in the thesis where they contribute to the development of the techniques formed.

2. TECHNIQUE FOR SIGNALS CONTAINING A SINGLE ACTIVE ELEMENT

In order to eliminate aliasing when a signal is uniformly sampled at a sub-Nyquist rate, it is first necessary to determine the cause of the aliasing in detail. This chapter is concerned with using a traditional model of a continuous analogue signal to investigate the origin of the aliasing phenomenon which occurs when a signal is sampled uniformly at below the Nyquist rate. The simplest possible signal, a sinusoid, is considered and a rigourous dealiasing algorithm is developed to allow ambiguities to be eliminated when the sinusoid is sampled by a system of sub-Nyquist samplers. The limitations of the algorithm are stipulated.

2.1 Fundamentals of Uniform Sampling.

Fourier's theorem states that any single valued periodic function, which has a repetition interval T_{ϕ}, can be represented by an infinite series of sine and cosine terms which are harmonics of the fundamental frequency, $f_{\phi} = 1/T_{\phi}$ **[3, Dunlop & Smith]**. It is therefore feasible to suppose that any analogue signal can be represented as a sum of sinusoids of different amplitude, frequency and phase. The magnitude of the analogue signal at time, t is modelled as $x_a(t)$ given by,

$$x_a(t) = \sum_{m=1}^{M} A_m \cos\left(2\pi . f_m t + \theta_m\right) \qquad (2.1)$$

where M denotes the number of frequency components. A_m, f_m and θ_m represent the amplitude, frequency and phase respectively of each component. To process this continuous-time signal by digital signal processing techniques, it is necessary to convert the signal into a sequence of instantaneous values by sampling it periodically every T_s seconds (uniform sampling at a rate $f_s = 1/T_s$) to produce a discrete-time signal, $x_d(n)$.

$$x_d(n) = x_a(nT_s) = \sum_{m=1}^{M} A_m \cos\,(2\pi.f_m.nT_s + \theta_m) \qquad (2.2)$$

where n is a positive number. N consecutive samples, where N is ideally a power of 2, then serve as the input to an N-point discrete Fourier transform (DFT), giving rise to amplitude and phase spectra with N/2 frequency bins in steps of $1/NT_s$ hertz from 0 to $(N/2 - 1)/NT_s$ hertz **[4, Benjamin]**. The amplitude and phase spectra of $x_a(t)$ and $x_d(n)$ are required for comparison to investigate the ambiguities generated when sampling at a frequency less than twice the maximum frequency component of the incoming signal; ie. when sampling at a sub-Nyquist rate. In order to simplify this investigation, a signal containing a single active element (M = 1) is considered.

2.2 Aliased Single Active Element Analysis.

Consider the case in which M = 1; ie. the signal contains only one sinusoidal component, thus, from equation (2.1),

$$x_a(t) = A_x.\cos\,(2\pi.f_x t + \theta_x) \qquad (2.3)$$

Taking the continuous Fourier transform of this, to determine the signal spectrum, gives

$$X_a(f) = \int_{-\infty}^{\infty} A_x.\cos\,(2\pi.f_x t + \theta_x).e^{-j2\pi ft}\,dt \qquad (2.4)$$

$$= \frac{A_x}{2} \int_{-\infty}^{\infty} \left[e^{j(2\pi f_x t + \theta_x)} + e^{-j(2\pi f_x t + \theta_x)} \right].e^{-j2\pi ft}\,dt$$

from the Euler identity†. Expanding gives **[3, Dunlop & Smith]**,

---

†The Euler identity states that $e^{\pm j\phi} = \cos\phi \pm j\sin\phi$ hence, $2.\cos\phi = e^{j\phi} + e^{-j\phi}$.

$$X_a(f) = \frac{A_x}{2}\int_{-\infty}^{\infty} e^{j\theta_x}.e^{-j2\pi t(f - f_x)} + e^{-j\theta_x}.e^{-j2\pi t(f + f_x)}\, dt$$

$$= \frac{A_x}{2}.e^{j\theta_x}.\delta(f - f_x) + \frac{A_x}{2}.e^{-j\theta_x}.\delta(f + f_x)$$

therefore,

$$X_a(f) = \frac{A_x}{2}.(\cos\theta_x + j\sin\theta_x).\delta(f-f_x) + \frac{A_x}{2}.(\cos\theta_x - j\sin\theta_x).\delta(f+f_x) \qquad (2.5)$$

where $\delta(x)$ is the impulse function. Let

$$a = A_x.\cos\theta_x \quad \text{and} \quad b = -A_x.\sin\theta_x \qquad (2.6)$$

giving

$$\theta_x = \arctan(-b/a) \quad \text{and} \quad A_x = \sqrt{(a^2 + b^2)} \qquad (2.7)$$

Therefore, the Fourier transform of a single harmonic in the time domain, may by represented as a pair of complex conjugates in the frequency domain.

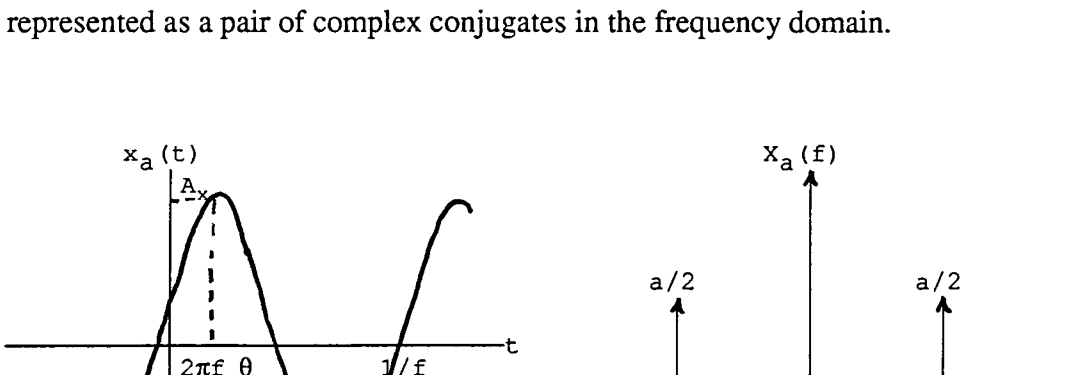


Figure 2.1. The time and frequency domain representations of a continuous sinusoidal signal. a & b are given in equation (2.6).

The sinusoid is sampled at regularly spaced intervals to produce the sequence of instantaneous values, $x_d(n)$. The uniform sampling process may be regarded as multiplying the continuous signal by a periodic series of impulses where each impulse corresponds to a sampling instance **[3, Dunlop & Smith]**. An infinite series of equidistant impulses, $u_0(t)$ may be represented by,

$$u_0(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_s) \tag{2.8}$$

A graphical representation of this is shown in figure 2.2.

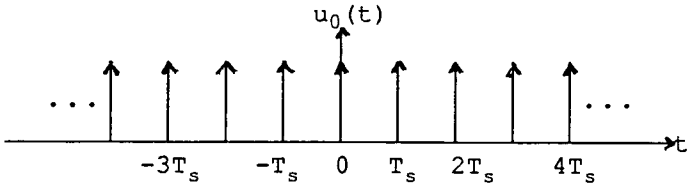


Figure 2.2. An infinite sequence of equidistant impulse functions.

The Fourier transform of a sequence of equidistant impulse functions is another sequence of equidistant impulses **[5, Brigham]**, given as,

$$U_0(f) = \frac{1}{T_s} \sum_{n=-\infty}^{\infty} \delta\left(f - \frac{n}{T_s}\right) \tag{2.9}$$

From the Convolution theorem, multiplication in the time domain translates to convolution in the frequency domain. Convolution of a function with $U_0(f)$ results in replication in the frequency domain. Hence, the spectrum at the output of a sampler whose input signal contains a single active element of amplitude A_x, phase θ_x, and frequency f_x, consists of pairs of sidebands spaced away by f_x from the sampling frequency harmonics $\pm f_s$, $\pm 2f_s$, $\pm 3f_s$, ... and 0; where $f_s = 1/T_s$ and the sideband pairs form the complex conjugate pairs $A_x/2.(\cos\theta_x + j\sin\theta_x)$ and $A_x/2.(\cos\theta_x - j\sin\theta_x)$.

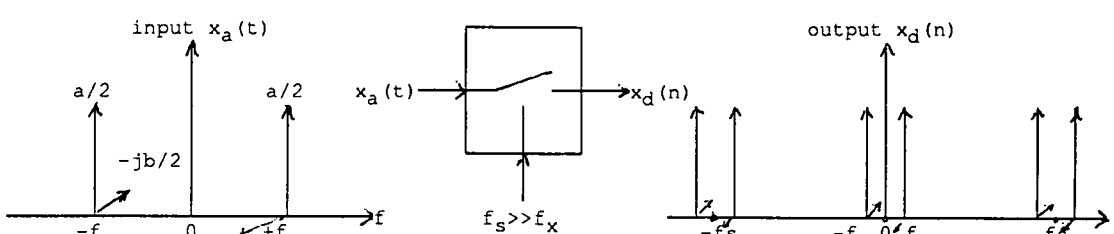


Figure 2.3. The spectra of the sinusoidal signal before and after super-Nyquist uniform sampling. a & b are given in equation (2.6).

As f_x increases from zero, the upper sideband of one replication tends towards the lower sideband of another, eg. f_x and $f_s - f_x$, until the point when they meet and aliasing occurs. That is, when $f_x = f_s - f_x$; ie. $f_x = f_s/2$. Therefore, to prevent this aliasing, it is necessary to follow the sampler by an 'ideal' low-pass filter (LPF) with a cut-off at half the sampling frequency. In practice, only an approximation can be implemented, leading to errors. The effect of such errors are not considered in this thesis.

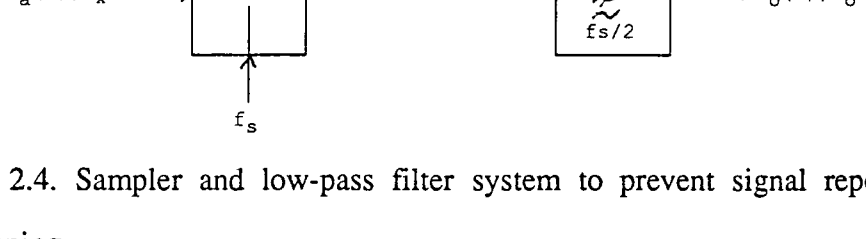


Figure 2.4. Sampler and low-pass filter system to prevent signal repetitions from overlapping.

The frequency of the analogue signal $x_o(t)$ output by the system in figure 2.4, $f_o = f_x$ for $f_x < f_s/2$. When $f_s > f_x > f_s/2$, $f_s - f_x$ will be less than $f_s/2$ and will therefore appear at the output. Furthermore, when $f_s < f_x < 1.5f_s$, $f_o = -f_s + f_x$ and when $1.5f_s < f_x < 2f_s$, $f_o = 2f_s - f_x$, and so on.

In general **[6, Underhill, Sarhadi & Aitchison]**, the frequency of the signal output by the filter, f_o is given by,

$$f_o = F - k.f_s \quad \text{for} \quad k.f_s < F < (k + 0.5).f_s$$

$$\text{and} \quad f_o = (k + 1).f_s - F \quad \text{for} \quad (k + 0.5).f_s < F < (k + 1).f_s \tag{2.10}$$

where k is a positive integer.

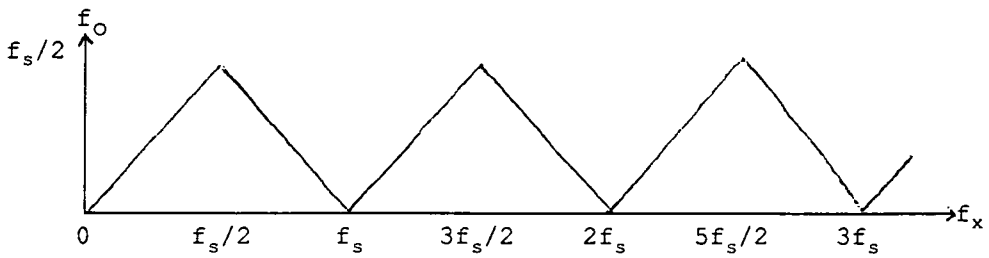


Figure 2.5. Frequency of signal output by sampler/LPF system against frequency of input sinusoid.

Hence, the analysis of a given f_o merely identifies the 'comb' of frequencies $(\pm f_x + k.f_s)$ hertz, where k is any positive integer and f_x is the frequency of the sinusoidal input of the sampler/LPF system giving rise to the output signal $x_o(t)$ with frequency, f_o. It can also be noted that as f_o increases with f_x, the signal sideband $(a + jb).\delta(f_x)$ appears at the output of the sampler/LPF system. As f_o decreases with f_x, the complex conjugate sideband $(a - jb).\delta(f_x)$ appears at the output. The phase of the sidebands will be shifted linearly by the characteristics of the 'ideal' LPF, but their amplitude will remain unchanged if the filter has an attenuation constant of unity in its pass-band. Thus, the frequency domain representation, $X_o(f)$ of the output signal, $x_o(t)$ is given by,

$$X_o(f) = (a + jb).\delta(f_x).L(f) \quad \text{for} \quad k.f_s < f_x < (k + 0.5).f_s$$

and $$X_o(f) = (a - jb).\delta(f_x).L(f) \quad \text{for} \quad (k + 0.5).f_s < f_x < (k + 1).f_s \qquad (2.11)$$

where k is a positive integer and L(f) represents the system transfer function of an ideal low-pass filter.

Therefore, for a signal consisting of a single active element, if the frequency, f_x of the sinusoid can be determined and the characteristics of the low-pass filter are known, then the amplitude and phase may be calculated from the output of the sampler/filter system. A method, although incomplete, to determine the frequency without ambiguity is outlined by **[6, Underhill, Sarhadi & Aitchison]**. A system of sub-Nyquist samplers will now be considered in order to determine the frequency, f_x of the sinusoidal input.

2.3 Proposed Dealiasing System.

An incoming signal of frequency f_x is sampled simultaneously by three samplers sampling at frequencies f_{s1}, f_{s2} and f_{s3}, with each followed by an 'ideal' low pass filter with a cut-off frequency $f_{s1}/2$, $f_{s2}/2$ and $f_{s3}/2$ respectively. The sampling frequencies are such that

$$0 < f_{s1} < f_{s2} < f_{s3} < 2.\text{signal bandwidth} \qquad (2.12)$$

The frequency of the output of each filter are represented by f_{o1}, f_{o2} and f_{o3}, whose values, in practice, are determined by following the filter with a digital frequency counter. Input circuitry is also necessary to ensure the signal contains only one active element and is bandwidth limited. The proposed single active element sub-Nyquist sampling system to resolve ambiguity is illustrated in figure 2.6 **[7, Sarhadi]**.

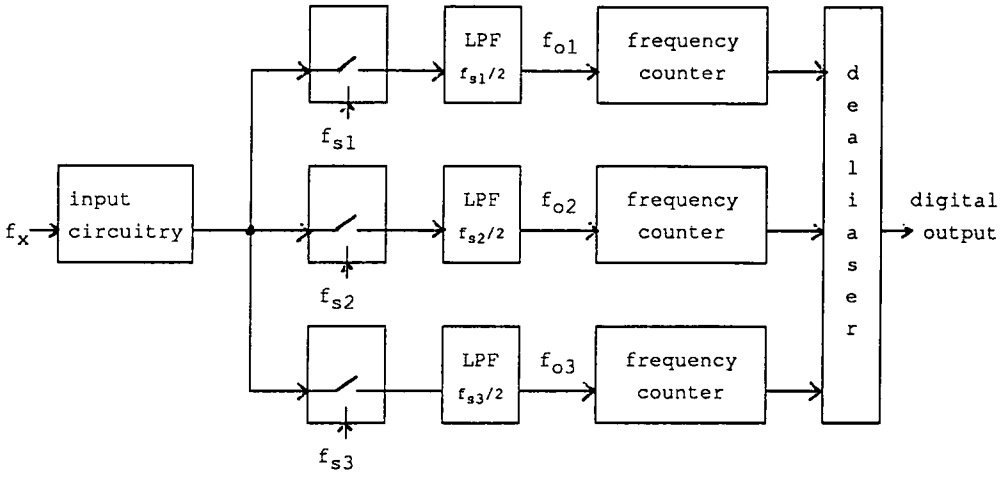


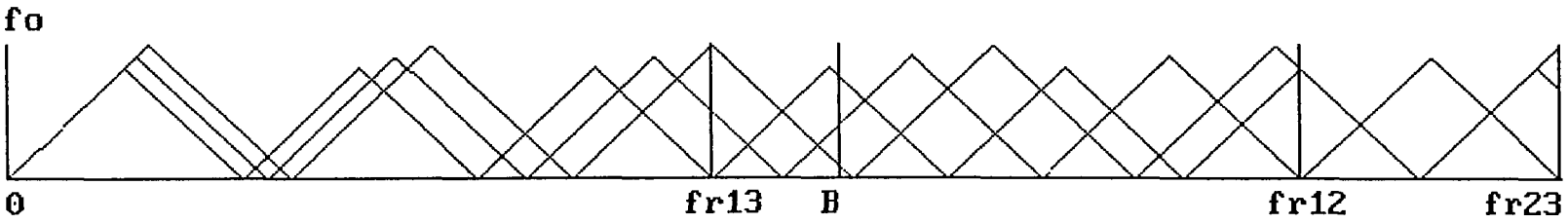
Figure 2.6. Proposed system for input containing a single active element.

The frequency at each output of the three low-pass filters varies with f_x as illustrated by figure 2.7 (from equation (2.10)). The frequency of the three sampler/filter outputs are described by equation (2.13).

$$
\begin{aligned}
f_{o1} &= f_x - p.f_{s1} && \text{for} && p.f_{s1} < f_x < (p + 0.5).f_{s1} \\
f_{o1} &= (p + 1).f_{s1} - f_x && \text{for} && (p + 0.5).f_{s1} < f_x < (p + 1).f_{s1} \\
f_{o2} &= f_x - q.f_{s2} && \text{for} && q.f_{s2} < f_x < (q + 0.5).f_{s2} \\
f_{o2} &= (q + 1).f_{s2} - f_x && \text{for} && (q + 0.5).f_{s2} < f_x < (q + 1).f_{s2} \\
f_{o3} &= f_x - r.f_{s3} && \text{for} && r.f_{s3} < f_x < (r + 0.5).f_{s3} \\
f_{o3} &= (r + 1).f_{s3} - f_x && \text{for} && (r + 0.5).f_{s3} < f_x < (r + 1).f_{s3}
\end{aligned}
\qquad \ldots(2.13)
$$

where p, q and r are positive integers.

The three sampler/LPF systems produce values for the three variables f_{o1}, f_{o2} and f_{o3} from the input of frequency f_x. A dealiasing algorithm is required to reproduce the value of f_x from the three frequencies f_{o1}, f_{o2} and f_{o3}. In order to do this without ambiguity, the three variables f_{o1}, f_{o2} and f_{o3} must take a unique combination of values for each possible input frequency f_x. The operational bandwidth of the system will be the maximum frequency of f_x which results in a combination of values of the variables f_{o1}, f_{o2} and f_{o3} that is not produced for any lower input frequency. Having established the limits of f_x, the dealiasing algorithm must solve equation (2.13) for f_x, given the values for f_{o1}, f_{o2} and f_{o3}.



fs1 = 10.0 fs2 = 11.0 fs3 = 12.0 => Operational frequency range = 35.5
fr12 = 55.0 fr23 = 66.0 fr13 = 30.0

Figure 2.7. Low pass filter output, example

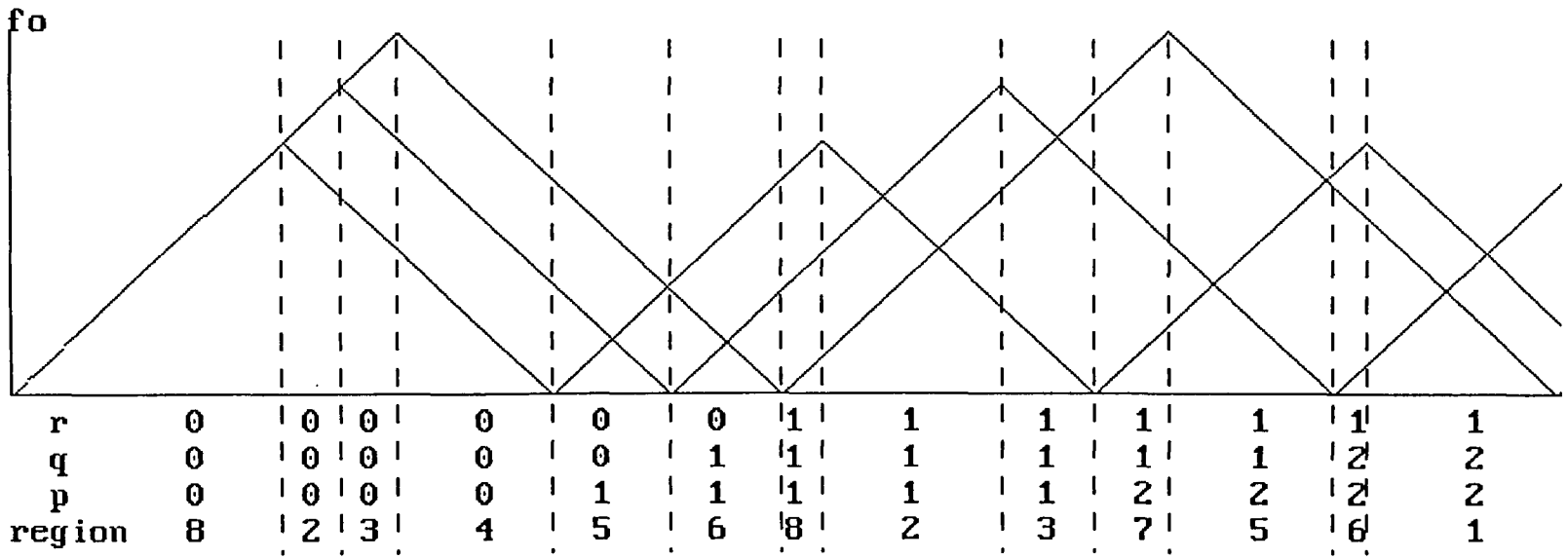


Figure 2.8. Arbitrary LPF output containing all eight possible patterns

2.4 Operational Bandwidth.

From figure 2.7, it can be seen that a point of symmetry, corresponding to an input frequency FR_{12}, exists in the output patterns generated by f_{o1} and f_{o2}, and is given by,

$$FR_{12} = n.f_{s1} = (n - 0.5).f_{s2}$$

for one n = 1, 1.5, 2, 2.5, ... Let $dfs_1 = f_{s2} - f_{s1}$. From $n.f_{s1} = (n - 0.5).f_{s2}$ we get $n = f_{s2}/(2.dfs_1)$. Therefore, the folding frequency for the pattern is,

$$FR_{12} = \frac{f_{s1}.f_{s2}}{2.dfs_1} \qquad (2.14)$$

Similarly, points of symmetry exists at input frequencies FR_{23} and FR_{13}, in the output patterns produced by f_{o2} & f_{o3} and f_{o1} & f_{o3} respectively, where,

$$FR_{23} = \frac{f_{s2}.f_{s3}}{2.dfs_2} \quad \text{and} \quad FR_{13} = \frac{f_{s1}.f_{s3}}{2.dfs_3} \qquad (2.15)$$

with $dfs_2 = f_{s3} - f_{s2}$, and $dfs_3 = f_{s3} - f_{s1}$.

For the dealiasing algorithm to identify a single frequency component, it is necessary for every possible frequency input to give a different output combination of f_{o1}, f_{o2} and f_{o3}; ie. a unique combination of f_{o1}, f_{o2} and f_{o3} must exist for every input frequency. If this was not the case, the input to the dealiasing algorithm (ie. f_{o1}, f_{o2} and f_{o3}) for one frequency component input would not differ from that for others, and so the algorithm would generate all possible frequencies that could produce such an input, and ambiguities would remain.

The dealiasing algorithm proposed in **[6, Underhill, Sarhadi & Aitchison]** states that "FR_{12} is taken to be the working frequency range of the whole system." However, this cannot be the case, as FR_{13} must be the smallest of FR_{12}, FR_{23} and FR_{13}, and, in some cases, the input to the dealiasing algorithm (ie. f_{o1}, f_{o2} and f_{o3}) will be the same for two distinct input frequencies.

Illustrative Example 2.1.

Consider the three samplers operating at frequencies with only 2 Hz difference, such that $f_{s1} = 1000$ Hz, $f_{s2} = 1002$ Hz and $f_{s3} = 1004$ Hz. The working frequency range claimed, $FR_{12} = 250{,}500$ Hz. If, for example, the input has a frequency of either 123,496 Hz or 127,504 Hz (both considerably less than FR_{12},) then $f_{o1} = 496$ Hz, $f_{o2} = 250$ Hz and $f_{o3} = 4$ Hz (from equation (2.13)) and the dealiasing algorithm will not be able to distinguish between the two possible inputs. Similarly if the input has a frequency of either 11,272 Hz or 239,728 Hz, for which $f_{o1} = 272$ Hz, $f_{o2} = 250$ Hz and $f_{o3} = 228$ Hz.

Consider any two sampler/filter outputs f_{oa} and f_{ob} (ie. f_{oa} and f_{ob} can be any pair of f_{o1}, f_{o2} and f_{o3}.) For most, but not all, frequency inputs below the folding frequency FR_{ab}, the output combination of f_{oa} and f_{ob} will be unique, but will be repeated for all inputs with a frequency greater than FR_{ab}. For example, an input frequency, f_1 which is greater than $f_{sa}/2$ such that $f_{oa} = f_{ob}$ will generate the same output as an input frequency, $f_2 = f_{oa}$ which is less than $f_{sa}/2$. For an input frequency slightly greater than FR_{ab}, say by δf, the output combination of f_{oa} and f_{ob} would not differ from that generated by an input of frequency $FR_{ab} - \delta f$. However, the output of a third sampler/filter, say f_{oc}, would differ for input frequencies up to, but not including, the point when $\delta f = f_{sc}/2$; and for all input frequencies below FR_{ab} for which the output combination of f_{oa} and f_{ob} was not unique. A unique output combination of f_{oa}, f_{ob}, f_{oc}, is thus produced for input frequencies below $FR_{ab} + f_{sc}/2$. Therefore, the optimum operational bandwidth is given by the minimum of $FR_{12} + f_{s3}/2$, $FR_{23} + f_{s1}/2$, and $FR_{13} + f_{s2}/2$. But it is known

that $f_{s1} < f_{s2} < f_{s3}$ (dfs_3 is greater than dfs_1 and dfs_2), and so $FR_{13} + f_{s2}/2$ is always the minimum of the three values . Thus the optimum operational bandwidth is given by,

$$B_{opt} = \frac{f_{s1}.f_{s3}}{2.(f_{s3} - f_{s1})} + \frac{f_{s2}}{2} \qquad (2.16)$$

Note that the optimum operational bandwidth is inversely proportional to the difference between the highest and lowest sampling frequencies. Clearly very wide bandwidth signals could be analysed if the difference in sampling frequencies is small. If f_{s1}, f_{s2} and f_{s3} are approximately equal (say, $\approx f_s$) and the difference between one sampling frequency and the next is approximately equal (say, $\approx$ dfs), then $dfs_1 \approx dfs_2 \approx$ dfs, $dfs_3 \approx$ 2.dfs and the operational bandwidth will be approximately,

$$B_{opt} \approx \frac{f_s^2}{4.dfs} + \frac{f_s}{2}$$

The is approximately half the bandwidth claimed by **[7, Sarhadi]**.

2.5 Single Active Element Dealiasing Algorithm.

The analysis of the output f_{o1} identifies the 'comb' of frequencies $(\pm f_x + p.f_{s1})$ hertz. The output f_{o2} analysis produces a similar ambiguity pattern $(\pm f_x + q.f_{s2})$ hertz. The two ambiguity patterns coincide on the true frequency, f_x and at a possible 'ghost' frequency. The correct input frequency can be identified from these two coincident frequencies by comparison with a third ambiguity pattern $(\pm f_x + r.f_{s3})$ hertz generated by f_{o3}, and can be determined by solving for f_x in the equations of (2.13).

The following dealiasing algorithm considers the patterns of the outputs f_{o1}, f_{o2} and f_{o3} simultaneously in order to solve the equations of (2.13) for f_x, given that f_x is less than B_{opt}. From figure 2.8, eight distinct regions can be identified in the output containing f_{o1}, f_{o2} and f_{o3}, corresponding to the three f_o's either increasing or decreasing with an increase in f_x.

Region 1. f_{o1} & f_{o3} decreasing and f_{o2} increasing with an increase in f_x.

$$f_{o1} = (p + 1).f_{s1} - f_x$$

$$f_{o2} = f_x - q.f_{s2}$$

$$f_{o3} = (r + 1).f_{s3} - f_x \qquad p = q = r + 1$$

Region 2. f_{o1} decreasing and f_{o2} & f_{o3} increasing with an increase in f_x.

$$f_{o1} = (p + 1).f_{s1} - f_x$$

$$f_{o2} = f_x - q.f_{s2}$$

$$f_{o3} = f_x - r.f_{s3} \qquad p = q = r$$

Region 3. f_{o1} & f_{o2} decreasing and f_{o3} increasing with an increase in f_x.

$$f_{o1} = (p + 1).f_{s1} - f_x$$

$$f_{o2} = (q + 1).f_{s2} - f_x$$

$$f_{o3} = f_x - r.f_{s3} \qquad p = q = r$$

Region 4. f_{o1}, f_{o2} and f_{o3} decreasing with an increase in f_x.

$$f_{o1} = (p + 1).f_{s1} - f_x$$

$$f_{o2} = (q + 1).f_{s2} - f_x$$

$$f_{o3} = (r + 1).f_{s3} - f_x \qquad p = q = r$$

Region 5. f_{o1} increasing and f_{o2} & f_{o3} decreasing with an increase in f_x.

$$f_{o1} = f_x - p.f_{s1}$$

$$f_{o2} = (q + 1).f_{s2} - f_x$$

$$f_{o3} = (r + 1).f_{s3} - f_x \qquad p = q + 1 = r + 1$$

Region 6. f_{o1} & f_{o2} increasing and f_{o3} decreasing with an increase in f_x.

$$f_{o1} = f_x - p.f_{s1}$$

$$f_{o2} = f_x - q.f_{s2}$$

$$f_{o3} = (r + 1).f_{s3} - f_x \qquad p = q = r + 1$$

Region 7. f_{o1} & f_{o3} increasing and f_{o2} decreasing with an increase in f_x.

$$f_{o1} = f_x - p.f_{s1}$$

$$f_{o2} = (q + 1).f_{s2} - f_x$$

$$f_{o3} = f_x - r.f_{s3} \qquad p = q + 1 = r + 1$$

Region 8. f_{o1}, f_{o2} & f_{o3} increasing with an increase in f_x.

$$f_{o1} = f_x - p.f_{s1}$$

$$f_{o2} = f_x - q.f_{s2}$$

$$f_{o3} = f_x - r.f_{s3} \qquad p = q = r \qquad \ldots(2.17)$$

Each of the above eight sets of simultaneous equations can be solved separately for p, giving,

Region 1.

$$p = \frac{f_{s1} - f_{o1} - f_{o2}}{dfs_1} = \frac{f_{o2} + f_{o3}}{dfs_2} = \frac{f_{s1} - f_{o1} + f_{o3}}{dfs_3}$$

Region 2.

$$p = \frac{f_{s1} - f_{o1} - f_{o2}}{dfs_1} = \frac{f_{o2} - f_{o3}}{dfs_2} = \frac{f_{s1} - f_{o1} - f_{o3}}{dfs_3}$$

Region 3.

$$p = \frac{f_{o2} - f_{o1}}{dfs_1} - 1 = \frac{f_{s2} - f_{o2} - f_{o3}}{dfs_2} = \frac{f_{s1} - f_{o1} - f_{o3}}{dfs_3}$$

Region 4.

$$p = \frac{f_{o2} - f_{o1}}{dfs_1} - 1 = \frac{f_{o3} - f_{o2}}{dfs_2} - 1 = \frac{f_{o3} - f_{o1}}{dfs_3} - 1$$

Region 5.

$$p = \frac{f_{o1} + f_{o2}}{dfs_1} = \frac{f_{o3} - f_{o2}}{dfs_2} = \frac{f_{o1} + f_{o3}}{dfs_3}$$

Region 6.

$$p = \frac{f_{o1} - f_{o2}}{dfs_1} = \frac{f_{o2} + f_{o3}}{dfs_2} = \frac{f_{o1} + f_{o3}}{dfs_3}$$

Region 7.

$$p = \frac{f_{o1} + f_{o2}}{dfs_1} = \frac{f_{s2} - f_{o2} - f_{o3}}{dfs_2} + 1 = \frac{f_{s3} - f_{o3} + f_{o1}}{dfs_3}$$

Region 8.

$$p = \frac{f_{o1} - f_{o2}}{dfs_1} = \frac{f_{o2} - f_{o3}}{dfs_2} = \frac{f_{o1} - f_{o3}}{dfs_3} \qquad \ldots(2.18)$$

For a value of p to be valid from any region, all three equations must yield the same value, and p, by definition, must be an integer. Furthermore,

$$f_x = (p + 1).f_{s1} - f_{o1} \qquad \text{for regions 1 to 4}$$

and $$f_x = f_{o1} - p.f_{s1} \qquad \text{for regions 5 to 8,} \qquad (2.19)$$

Therefore, a value of p will only be valid if it also gives a value for the correct coincident frequency, f_x, as greater than or equal to 0 and less than B_{opt}. By careful observation of the equations of (2.18) and noting that the input frequency can fall in only one of the eight regions at any time, it can be seen that only one such value of p is ever produced. The frequency ambiguity is thus eliminated.

2.6 Errors Imposed by the Limitations of the DFT.

The frequency counters used in the proposed dealiasing system of section 2.3 can be replaced by spectral analysers that would not only give the frequency of each sampler/LPF output, but also the amplitude. This could then be used with equation (2.11) to determine the amplitude of the sinusoidal input. Each spectral analyser will need to perform a Fourier analysis of each sampler/LPF output which requires the use of the discrete Fourier transform. There is a limited resolution to which an N-point discrete Fourier transform can determine the frequency of an active element of a signal. Any active element must be represented spread over the coefficients of the DFT and assuming that most of the energy is concentrated in the single nearest coefficient, the maximum frequency error in the spectral analysis is given by **[7, Sarhadi]**,

$$|dfo_{max}| = \frac{f_s}{2.N} \qquad (2.20)$$

where f_s is the sampling frequency. The effects of this error on the dealiasing algorithm must be determined.

The frequency error of equation (2.20) leads to an error in calculating the cycle count, p, when using the equations of (2.18).

Consider region 1.

$$p = \frac{f_{s1} - f_{o1} - f_{o2}}{dfs_1} = \frac{f_{o2} + f_{o3}}{dfs_2} = \frac{f_{s1} - f_{o1} + f_{o3}}{dfs_3}$$

The error in each f_o will produce an error in p for each of the three sections of the equation above, giving,

$$p + dp_1 = \frac{f_{s1} - (f_{o1} \pm dfo_1) - (f_{o2} \pm dfo_2)}{dfs_1}$$

$$p + dp_2 = \frac{(f_{o2} \pm dfo_2) + (f_{o3} \pm dfo_3)}{dfs_2}$$

$$p + dp_3 = \frac{f_{s1} - (f_{o1} \pm dfo_1) + (f_{o3} \pm dfo_3)}{dfs_3}$$

Hence, substituting for dfo_i using equation (2.20) and assuming that each DFT uses the same N number of points (valid if the differences in the sampling frequencies are small,)

$$|dp_1| = \frac{f_{s1} + f_{s2}}{2.N.dfs_1}, \quad |dp_2| = \frac{f_{s2} + f_{s3}}{2.N.dfs_2}, \quad |dp_3| = \frac{f_{s1} + f_{s3}}{2.N.dfs_3} \qquad \ldots(2.21)$$

Similarly for the other seven regions. The value of p that is to be used in calculating f_x from equation (2.19) must be the one containing least error. As $f_{s1} < f_{s2} < f_{s3}$, dp_3 is always the minimum of dp_1, dp_2 and dp_3. Therefore, by including dp_3 with p in equation (2.19),

$$f_x + |dfx_{max}| = (p + |dp_3| + 1).f_{s1} - f_{o1} + |dfo_{1max}|$$
$$= f_x + |dp_3|.f_{s1} + |dfo_{1max}|$$

Hence,

$$|dfx_{max}| = |dfo_{1max}| + |dp_3|.f_{s1} = \frac{f_{s1}.f_{s3}}{N.(f_{s3} - f_{s1})} \qquad (2.22)$$

The errors that can occur in both p and f_x, mean that the dealiasing algorithm must be refined. For a value of p to be valid from any of the eight regions, each of the three equations must yield a value which is equal within the error bands dp_1, dp_2 and dp_3 respectively. Furthermore, p must not differ from an integer value by more than dp_1,

dp_2 or dp_3 for each of the three equations, and the value of $(f_x + dfx)$ generated from p must be less than,

$$B_{err} + |dfx_{max}| = \frac{f_{s1}.f_{s3}}{2.(f_{s3} - f_{s1})} \tag{2.23}$$

Where B_{err} is the operational bandwidth of the system when DFT errors are also considered, given by,

$$B_{err} = \frac{f_{s1}.f_{s3}}{(f_{s3} - f_{s1})}.\left[\frac{1}{2} - \frac{1}{N}\right] \tag{2.24}$$

for $N > 2$.

B_{err} is slightly less than B_{opt} (for $f_{s2} << FR_{13}$ and assuming $N >> 2$) as it is no longer possible to guarantee that the output combination f_{o1}, f_{o2}, f_{o3} will be unique for input frequencies greater than FR_{13}.

The frequency resolution with which the input signal can be analysed,

$$R_i = 2.|dfx_{max}| = \frac{2.f_{s1}.f_{s3}}{N.(f_{s3} - f_{s1})} \tag{2.25}$$

By rearranging equation (2.25) and substituting into equation (2.24),

$$B_{err} = R_i.N.\left[\frac{1}{2} - \frac{1}{N}\right] \tag{2.24}$$

Hence,

$$N = \frac{4.B_{err}}{R_i} + 2 \tag{2.26}$$

This clearly shows that the computational workload of the technique is inversely proportional to the frequency resolution.

Furthermore, for the modified algorithm to perform correctly, it is necessary for dp_1, dp_2 and dp_3 to be less than 0.5. This requirement is satisfied only if (from equation (2.21)),

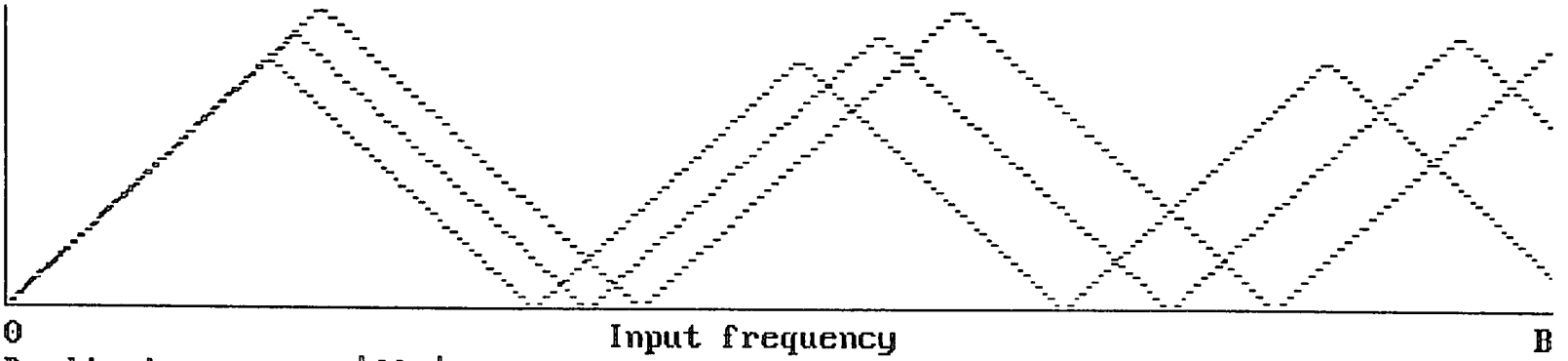
$$N > \frac{f_{s2} + f_{s1}}{f_{s2} - f_{s1}} \quad \text{and} \quad N > \frac{f_{s3} + f_{s2}}{f_{s3} - f_{s2}} \tag{2.27}$$

A 'C' simulation of this algorithm is presented in appendix A. The implementation rigourously defines the algorithm but when executed shows that the algorithm is not always error free! The rare errors observed are believed to occur due to floating-point arithmetic errors or the assumption that all three DFTs use an equal number of consecutive samples no longer being valid (which is the case for small N or large differences in the sampling frequencies.) As N increases, no errors can be found.

Figure 2.9 shows an example of the ideal sampler/filter output patterns generated when DFT errors from an 85-point transform are considered, and the error made by the dealiasing algorithm in attempting to determine the frequency of the signal input. The error, $|dfx|$ is the absolute difference between the actual frequency of the signal input and its frequency as evaluated by the algorithm. The magnitude of the error varies in an approximately triangular fashion, increasing and decreasing as the estimated outputs f_{o1}, f_{o2} and f_{o3} tend away and towards the correct outputs of f_{o1}, f_{o2} and f_{o3} (given when N is infinite) respectively. The complex nature in which the error changes is not of great importance. However, note that the level of error is consistently below the maximum permissible error, $|dfx_{max}|$. Therefore, the frequency of the input sinusoidal signal can be found within a calculated tolerance when the frequencies of the sampler/filter outputs are known with error.

fs1 = 10.000 fs2 = 11.000 fs3 = 12.000
Operational frequency range, B = 29.294118
Number of points in DFTs = 85
Maximum permissible error, |dfx_max| = 0.705882

Sampler/Filter outputs: fo1, fo2 and fo3



Dealiasing error: |dfx|

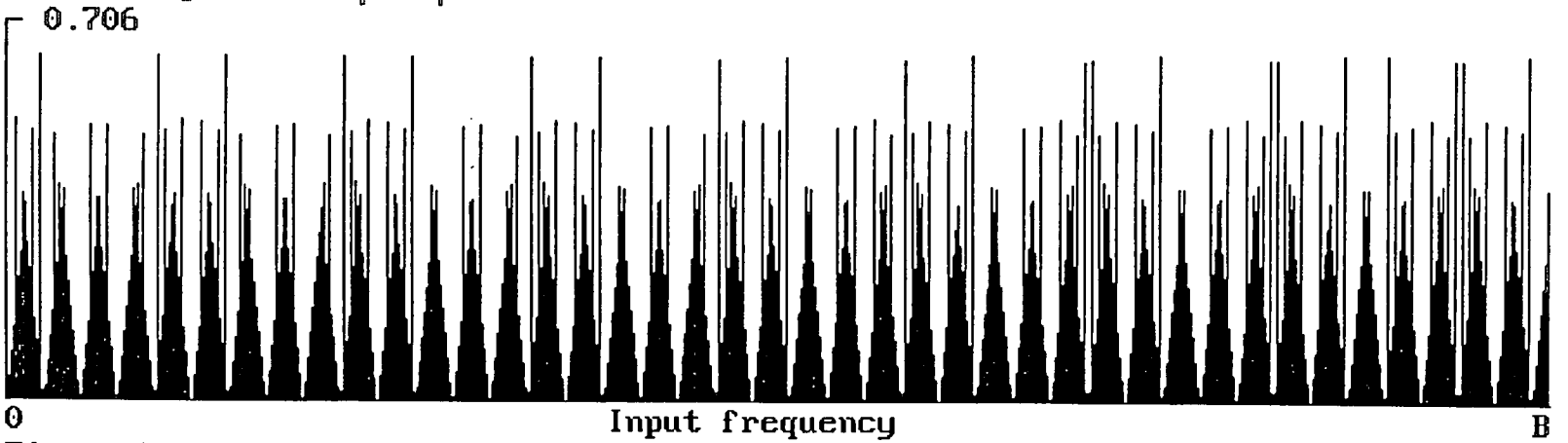


Figure 2.9. Dealiasing algorithm errors below permissible maximum, example

2.7 Summary.

A theoretical method has been established to eliminate not only the frequency ambiguity, but also any amplitude ambiguity, resulting from sampling a pure sinusoidal signal below the Nyquist rate. The dealiasing algorithm developed considers the output of three sampler/LPF systems to determine the frequency of the sinusoidal input. There is a limit to the maximum frequency of the signal in relation to the three sampling frequencies used, which represents the operational bandwidth, B_{opt} (given by equation (2.16)) of the technique.

In practice, the limitations of an N-point discrete Fourier transform mean that the frequency of the signals at the output of each sampler/LPF filter can only be specified within some known tolerance. This error results in the technique being able to guarantee an unambiguous output with a small error (the maximum permissible error is given as $|dfx_{max}|$ by equation (2.22)) at a slightly reduced bandwidth, B_{err} (given by equation (2.24).)

The technique is only applicable to pure sinusoidal signals and needs to be extended to apply to signals capable of containing any number of harmonics (signals consisting of multiple active elements.)

3. TECHNIQUE WITH UNIFORM SAMPLING FOR SPECTRAL ANALYSIS

The dealiasing technique described in chapter 2 is to be extended for signals consisting of multiple active elements. Each active element in the input signal of the system (shown in figure 2.6) will result in an aliased spectral line in the spectrum of the signal at the output of each sampler/LPF system. If the frequency counters used in the proposed dealiasing system of section 2.3 are replaced by spectral analysers, as in section 2.6, then the spectrum of the signal at the output of the sampler/LPF systems can be determined. These spectra must be analysed to determine the frequency of each component of the input signal.

3.1 Multiple Active Element Ambiguity Reduction Algorithm $N^{O}\cdot 1$.

The technique developed to analyse an aliased single active element can be extended to identify, the components of an aliased multiple active element signal. Each active element results in a frequency component in the output of each of the three sampler/filter systems, some of which could overlap. There is, therefore, a spectrum of aliased lines at each sampler/filter output; namely, FFT1, FFT2, and FFT3. The proposed technique to resolve the frequency ambiguities involves the use of equation (2.19) to identify a possible input frequency in the manner described for the single active element analysis, for every combination of the aliased lines in FFT1, FFT2 and FFT3.

3.2 Multiple Active Element Ambiguity Reduction Algorithm $N^{O}\cdot 2$.

An alternative to the algorithm $N^{O}\cdot 1$ described in section 3.1, is the following simple iterative algorithm which yields the same spectrum as algorithm $N^{O}\cdot 1$. For any aliased line in FFT1, say e_1, a set of possible values for the frequency components of the original signal exists; ie. each aliased line in the sample/filter output can identify a

'comb' of possible input harmonic frequencies. The frequency, sf_1, of the members of this set, s_1, are given by,

$$sf_1 = N_1.f_{s1} \pm e_1 \quad \text{for all } e_1 \in FFT1 \tag{3.1}$$

where N_1 is a positive integer and $0 \leq sf_1 < B_{opt}$. Frequencies that do not exist in the set s_1 will not exist in the original signal. Similarly, two further sets, s_2 and s_3, can be produced for the aliased lines in FFT2 and FFT3 respectively. The frequency components of the original signal, among some 'ghost' values (the frequency of components erroneously identified as part of the original signal,) are the frequencies that form the intersection of the sets s_1, s_2, and s_3. Clearly, if a greater number of aliased spectra had been generated by additional samplers operating at different frequencies, more ambiguity 'ghosts' could be eliminated.

3.3 Illustration of Inherent Ambiguity.

Unfortunately, a fundamental problem is inherent with the approach made by both algorithms N^{o}·1 and N^{o}·2, described in sections 3.1 and 3.2 respectively. An illustration of this problem is given in the following example.

Illustrative Example 3.1.

Let f_{s1} = 1000 KHz, f_{s2} = 1001 KHz, and f_{s3} = 1002 KHz, giving an optimum operational bandwidth of 251,000.5 KHz from equation (2.16). Consider the case in which the original signal contains three active elements at 54,214 KHz, 150,920 KHz, and 191,782 KHz.

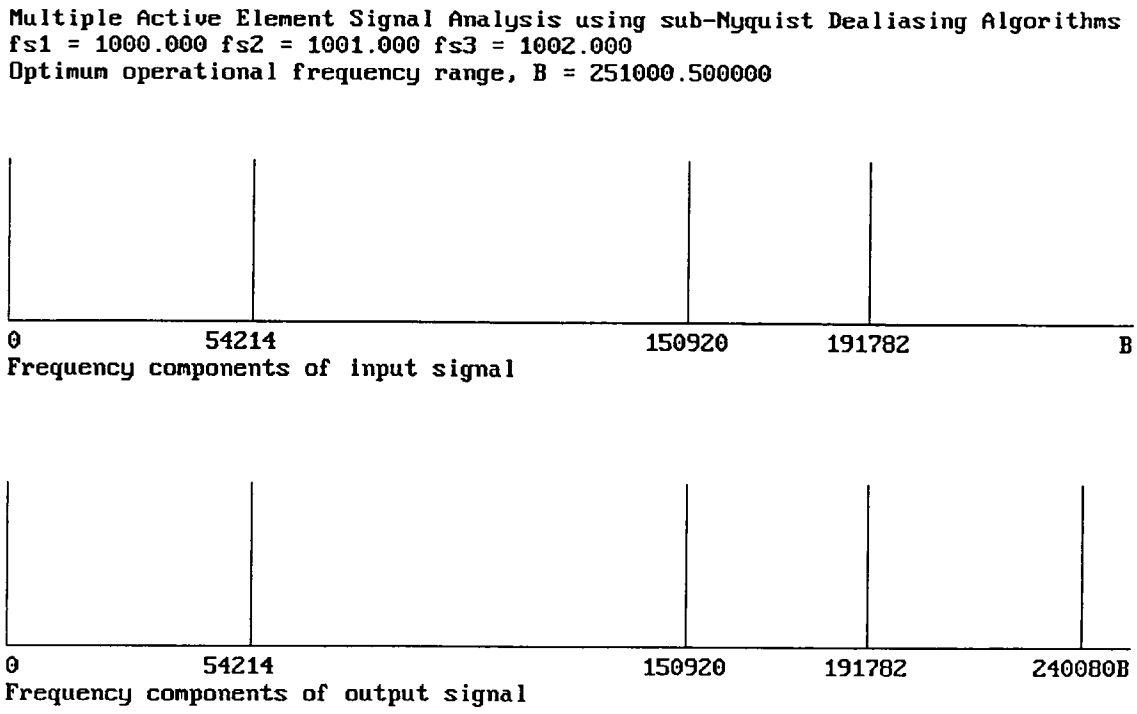


Figure 3.1. Illustrative example 3.1 of the inherent ambiguities remaining after execution of either multiple active element ambiguity reduction algorithm.

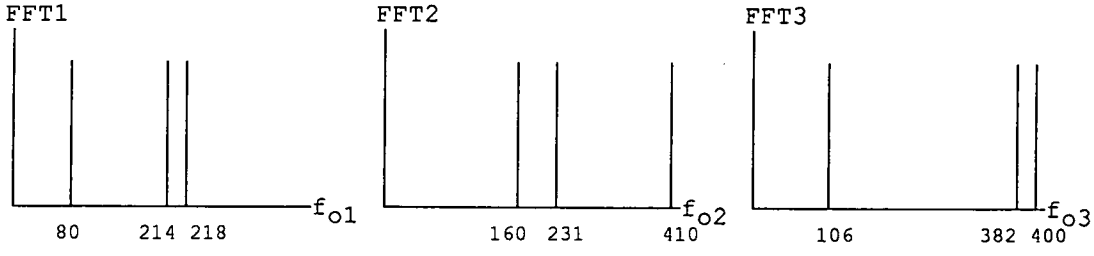


Figure 3.2. Frequency of signal harmonics output by each of the three sampler/LPF systems (for example 3.1).

If the original signal also contained an active element at 240,080 KHz, for which f_{o1} = 80 KHz, f_{o2} = 160 KHz, f_{o3} = 400 KHz, then the aliased spectra FFT1, FFT2, and FFT3 would not differ in the case above. Therefore, processing the three aliased spectra above for the original signal shown, as described by either algorithm, will yield the active elements of the original signal as 54,214 KHz, 150,920 KHz, 191,782 KHz and 240,080 KHz, although the true original signal contains only the first three of these components.

Figure 3.3 further illustrates this problem for a sparsely populated frequency spectrum. The input signal contains only four harmonics, yet the dealiasing algorithm erroneously identifies six harmonics.

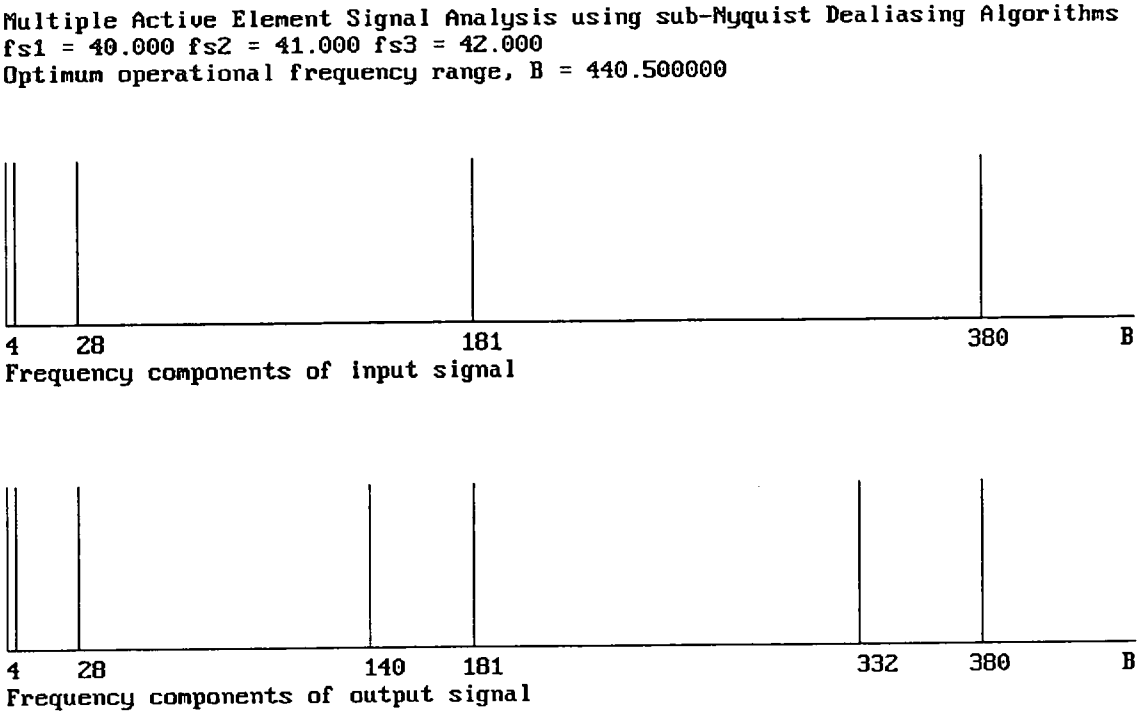


Figure 3.3. Illustrative example 3.2 of the inherent ambiguities remaining after execution of either multiple active element ambiguity reduction algorithm.

In most cases, each active element in the input signal will cause some change to at least one of the spectra of the sampler/filter outputs, and both algorithm $N^{O}\cdot 1$ and $N^{O}\cdot 2$ will be able to determine the frequencies of the input signal harmonics without ambiguity. However, there is the possibility that the algorithms will incorrectly identify harmonics as part of the input signal.

3.4 An Investigation of the Relative Efficiency of the Two Algorithms and a Full-Scale FFT.

Assume the algorithms are implemented on a modern, high speed, digital signal processor for which the number of clock cycles required to perform multiplication or addition are equal, and take the form as implemented in the simulation program. It is known that a radix-2 N-point decimation-in-frequency FFT imposes a workload of $(N/2).\log_2 N$ complex multiplications and $N.\log_2 N$ complex additions **[4, Benjamin]**. Any complex multiplication requires, at most, four real multiplications and two real additions. Also, any complex addition requires two real additions. Therefore, it is possible to say that an N-point FFT imposes a workload proportional to $5N.\log_2 N$.

Assume that the sub-Nyquist system contains k samplers all operating at a frequency approximately equal to f_s.

Consider a signal containing M active elements, with a bandwidth B requiring frequency analysis by use of a discrete Fourier transform with a frequency resolution R. Using a super-Nyquist system sampling at just more than 2B, a (B/R)-point FFT, at the bare minimum, would be required. This would impose a workload proportional to,

$$W_{FFT} = 5.(B/R).\log_2(B/R) \qquad (3.2)$$

For algorithm $N^{o}\cdot 1$ described in section 3.1, the worst case occurs when, for every one of the k aliased spectra produced, ie. FFT1, FFT2, ..., FFTk, there is a different ambiguity line for each of the M input signal components. Thus, in considering every possible combination of the aliased lines in the k, FFT spectra, the single active element dealiasing algorithm described in section 2.5 must be executed M^k times. Of these M^k executions, some combinations will not yield a possible input element, and a maximum of $2.M.B/f_s$ will be produced. Assume that the single active element dealiasing algorithm requires approximately 110 multiplications, additions and comparisons

(judged from the implementation in appendix B.) Therefore, the workload imposed by considering every line in the k aliased spectra is proportional to $110.M^k$. In addition, there are k, N-point FFTs that require computation, leading to a further workload proportional to $k.5.(4B/R + 2).\log_2(4B/R + 2)$. Therefore, the total computational overhead,

$$W_1 = 110.M^k + k.5.(4B/R + 2).\log_2(4B/R + 2)$$

$$\approx 10.(11.M^k + 4.k.(B/R)) + 4.k.W_{FFT} \qquad (3.3)$$

For algorithm N°·2 described in section 3.2, k sets need to be formed. In the worst case, the maximum cardinality of a set is $2.M.B/f_s$. One real multiplication and one real addition are needed to calculate each member and one list assignment is required. Assume that a list assignment requires twenty times more clock cycles than multiplication or addition (judged from the implementation in appendix B.) Therefore, the computational workload in calculating the k sets is proportional to $(20 + 1 + 1).k.2.M.B/f_s = 44.k.M.B/f_s$. The union of the sets is then required, which will take, at most, $2.M.B/f_s$ list assignments when there is a match for every member of one of the k sets, and $(k - 1).M.B/f_s$ comparisons. Therefore, the workload imposed in resolving the ambiguity spectra is proportional to $(45.k - 1).M.B/f_s$. In addition, there is the k, FFT computational workload proportional to $k.5.(f_s/R).\log_2(f_s/R)$, as before. Therefore, the total computational overhead,

$$W_2 = (45.k - 1).M.B/f_s + k.5.(4B/R + 2).\log_2(4B/R + 2)$$

$$\approx 5.B.k.(9/f_s + 8/R) + M.B/f_s + 4.k.W_{FFT} \qquad (3.4)$$

Clearly, algorithm N°·1 would be a better method to use than algorithm N°·2 if $W_1 < W_2$; ie. if $110.M^k < (45.k - 1).M.B/f_s$. Consider the case in which three samplers are used (k = 3) and approximate, thus algorithm N°·1 should be used only if $M^2 < B/f_s$,

which will be true when the number of active elements squared is less than the number of alias folds. As the number of samplers increases, which is necessary to improve dealiasing, it becomes evident that algorithm $N^{O}\cdot 2$ is by far the better.

Figure 3.4 shows just one case of how the proportional computational workloads W_{FFT}, W_1 and W_2 vary for an increasing number of harmonics in the input signal, M. Unfortunately, but not surprisingly, it is evident that the computational workload imposed by the two algorithm is far greater than that required by a full-scale FFT. The workload W_{fft} is so small relative to workloads W_1 and W_2 that it appears to run along the abscissa. For the overheads to be kept to a minimum, it is necessary to reduce the number of samplers k. However, k must be increased to resolve the ambiguities of a signal with a high active element population.

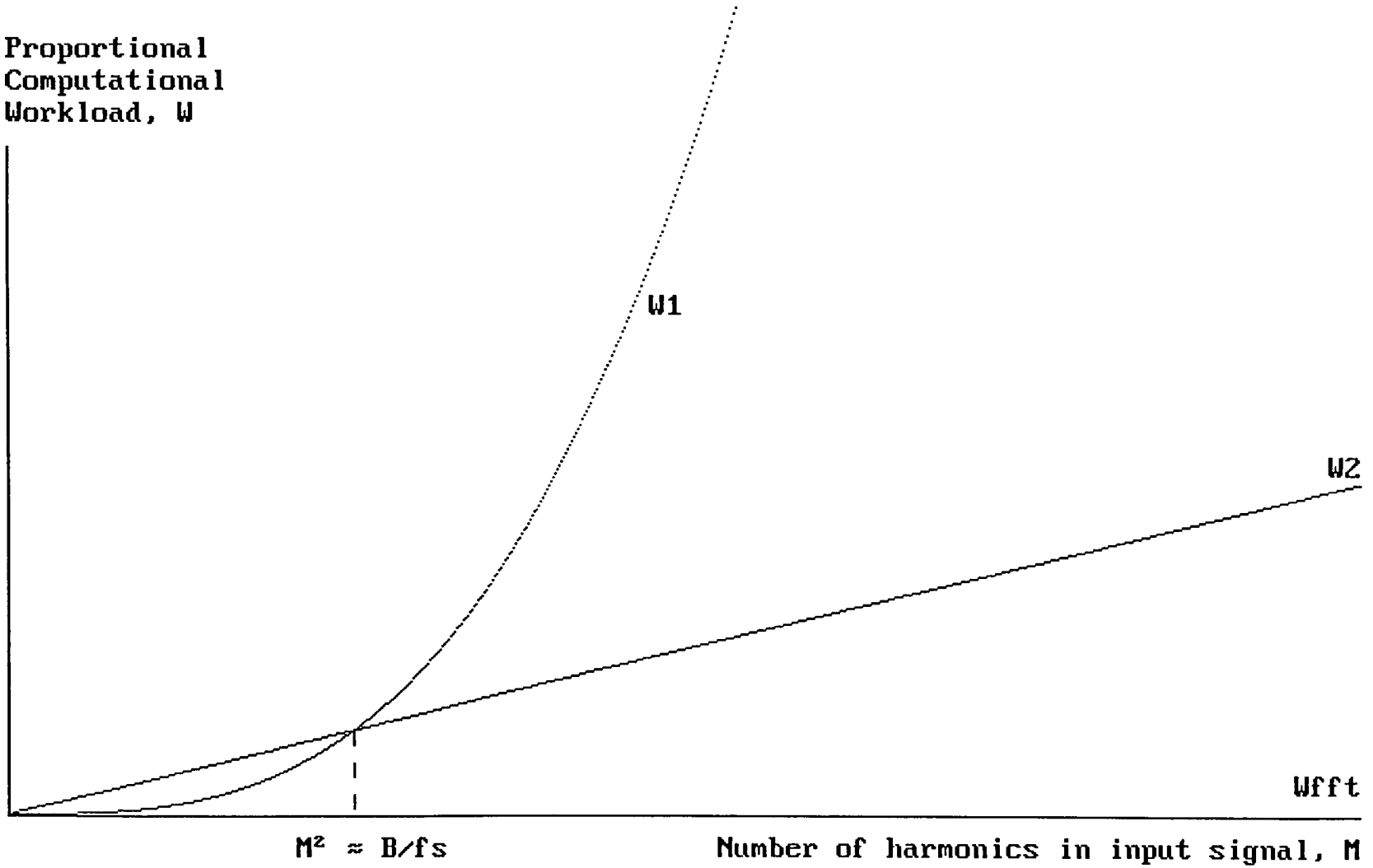


Figure 3.4. Workload involved with k = 3, B = 2.00e+006, R = 200.00, fs ≈ 150.00

3.5 Optimising Parameters to Minimise the Number of 'Ghost' Frequencies Possible in the Analysis of a Multiple Active Element Signal.

It has been illustrated in section 3.3 that the dealiasing technique developed is by no means perfect, but it does achieve a reduction in ambiguity. The aim is to select parameters so that the reduction is maximal and efficient in computation.

Consider the problem illustrated by example 3.1 in which an input of three active elements is incorrectly analysed as containing four elements. This is a direct result of the inability for the aliased spectra to change in response to the existence of a fourth potential element in the original signal. For only the frequency components of the original signal and no 'ghosts' to appear in the final output spectrum, it is necessary to ensure that the aliased spectra alter for every possible additional active element in the input.

For any given input active element there must be a unique combination of sampler output frequencies for that element to be identified without ambiguity. It has been shown that such identification is possible for a signal of bandwidth, B_{opt}. Let the unique combination for any frequency component of a multiple active element signal, output by a system containing k samplers, be represented by,

$$f_{o1}, f_{o2}, \ldots, f_{ok}.$$

To prevent 'ghost' frequencies appearing in the output spectrum, this combination, for a single input, must differ from any possible combination of other outputs. That is to say, at least one member of $\{f_{oi} | i=1 \text{ to } k\}$ in any given combination must not be repeated for any other of the possible unique combinations that could be produced. An arbitrary set of combinations for which this is true shall be represented by **T**. In a sub-Nyquist system, **T** must be a proper subset of the set of all practically possible output

combinations, **U**. However, **T** is any arbitrary set for which at least one fo_i is unrepeated for all its elements, and so, mathematically speaking, **T** may not be a subset of **U**.

If the bandwidth of the original signal is B and the signal is to be analysed with a finite frequency resolution R, then the number of frequency bins in the input is $\lfloor B/R \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer that is less than or equal to x. Each of k samplers sampling at a frequency f_{sj} will give inputs for an FFT whose output will contain $\lfloor f_{sj}/2R \rfloor$ frequency bins of interest (the others being complex conjugates.) The total number of combinations of the sampler outputs that are mathematically possible is,

$$(f_{s1}.f_{s2}. \ \ldots \ .f_{sk}) \ / \ (2R)^k = \text{cardinality of the universal set.}$$

However, in practice only $\lfloor B/R \rfloor$ combinations exist, corresponding to a unique output combination for each frequency bin in the input; ie.

$$\text{the cardinality of set } \mathbf{U} = \lfloor B/R \rfloor.$$

The maximum number of active elements which could possibly be correctly identified without ambiguity, ie. the maximum cardinality of **T**, is given from the mathematics of combinations and permutation as,

$$\text{max} = 2 + \sum_{j=1}^{k} (\lfloor f_{sj}/2R \rfloor - 2) \quad \text{for} \quad k > 0 \tag{3.5}$$

and the minimum cardinality of **T**, occurring for example when the input signal contains a harmonic for each and every frequency bin from 0 to the first folding frequency of the highest frequency sampler ie. $f_{s\_max}/2$, as,

$$\text{min} = \text{the greatest of } \lfloor f_{si} \ / \ 2R \rfloor_{i=1 \text{ to } k} \tag{3.6}$$

It can be seen that max = min = $\lfloor f_{s1}/2R \rfloor$ for k = 1, and that the cardinalities of **T** and **U** are equal when $f_{sk}/2 = B$; ie. the sub-Nyquist scheme tends to that of the super-Nyquist system as $f_{sk}/2$ tends to B when k = 1. Remember, however, that **T** is not necessarily a subset of **U**, and equation (3.5) talks only of cardinality. Therefore, only the probability of correct analysis may be increased by an increase in k or f_{sk}. An increase in f_{sk} defeats the aim of using sub-Nyquist sampling and an increase in k results in a greater computational workload. It is therefore necessary to increase the effective number of samplers, k without using excessive hardware. This might be achieved by using random sampling.

3.6 Summary.

Two algorithms have been proposed to greatly reduce, but not eliminate, the ambiguities produced by sampling a multiple active element signal at below the Nyquist rate with a number of samplers. As the technique results in only a reduction of ambiguities, a detailed error analysis is not presented. An increase in the number of samplers is required for further reduction of the frequency ambiguities. However, the computational workload imposed can be excessive for highly populated signals and increases dramatically with a increasing number of samplers. An improved algorithm is required which has fewer computational overheads and eliminates all frequency ambiguities for even highly populated signals.

4. SPECTRAL ANALYSIS WITH SUB-NYQUIST PSEUDO-RANDOM SAMPLING

4.1 Introduction.

If there are equal time intervals of $1/f_s$ between samples of a signal with a harmonic at frequency f_x, then the resultant output contains other harmonics at frequencies $f_s \pm f_x$, $2f_s \pm f_x$, $3f_s \pm f_x$, ... However, under certain conditions, if sampling points are formed at unequal intervals, this phenomenon disappears and the aliasing effect becomes absent **[8, Bilinsky, Vystavkin & Mikelson]**. The objective is to determine the requisites of the irregular sampling signal so that, even with sampling rates below the Nyquist level, the harmonics of the original signal may be determined with the minimum of error and computational overheads. The bandwidth limitation that must be imposed on the original signal also needs to be determined.

Uniform sampling has the limitation that aliasing occurs if the rate of sampling is below the Nyquist rate. It is expected that irregular sampling will have limitations if the maximum sampling rate is below the Nyquist level, but the limitation will not be an aliasing phenomenon. The limitations of irregular sampling are researched.

It is suggested by **[9, Beutler]** that random sampling of a wide-sense stationary stochastic process is alias-free and the error-free recovery of the process is possible if Poisson random sampling (sampling interval steps differ independently with identical exponential probability densities) is employed with an average sampling rate that does not fall below the Nyquist rate. **[10, Masry]** shows that such a Poisson random sampling scheme results in consistent alias-free estimates of the process' spectral density. Iterative methods have been developed which permit signal recovery with some additional uncorrelated background noise from "non-uniform samples with Poisson or uniform distributed epoches" and uniform samples with jitter or missing samples, in

[11, 12, Marvasti] and [13, Wiley], but these methods assume an average sampling rate higher than the Nyquist rate.

The papers [9, 10, 11, 12 and 13] make no comment on the bandwidth limitation to be imposed on the input signal, or the finite duration over which samples are taken. In a practical system, it is not possible to sample a non-band-limited signal over an infinite duration. Instead, sampling is limited to a finite number of sampling instances over a known duration, and is only applicable to band-limited signals. Furthermore, it can be inferred from the definition of Poisson random sampling (sampling interval steps differ independently with identical exponential probability densities [9, Beutler]) that the step between one sampling instance and the next, can be of any size greater than some preset minimum. In practice, however, there must also be a limit to the maximum possible step and so such an ideal scheme is not feasible. The exact instances that each sample is taken, although irregular, may be known from the generating function. With this additional information at hand and the practical limitations of the Poisson random sampling scheme noted, it is intended that a sampling scheme with a maximum rate that falls below the Nyquist rate may be defined which allows alias-free sampling of a signal.

What is meant by alias-free sampling of a signal when referring to samples taken at irregular intervals? When samples are taken at uniform intervals, sampling is said to be alias-free if the original signal can be unambiguously reconstructed from the samples. An irregular sampling scheme will be referred to as 'alias-free' if there is the capability to consistently reconstruct the spectrum of the original process from the spectral properties of the samples (which may be derived from the magnitude of the samples and the instances at which they are taken.)

A transform must be rigourously defined to find the spectral property of a signal sampled at irregularly spaced intervals, as the traditional discrete Fourier transform, defined for samples taken at uniformly spaced intervals, is no longer suitable. The

properties of the transform depend on the nature of the irregular sampling instances, and are investigated.

The irregularly spaced sampling instances are generated using two rigourously defined schemes. In both schemes, the step between one sampling instance and another will be one of a finite set of possible changes. The change selected will depend on the outcome of an independent pseudo-random number generator. For this reason, the transform will be referred to as the 'pseudo-random discrete Fourier transform.'

4.2 Theoretical Development of the Pseudo-random Discrete Fourier Transform.

The following derivation closely resembles that of the Discrete Fourier Transform for samples taken uniformly as described by **[5, Brigham]**.

Consider a band-limited, analogue signal $x_a(t)$, with a fundamental period $T_\emptyset$, represented as a sum of sinusoids of different amplitude, frequency and phase. Alternatively, represented as the Fourier series,

$$x_a(t) = \sum_{m=0}^{M-1} \left[a_m.\cos\ (2\pi.f_m.t) + b_m.\sin\ (2\pi.f_m.t) \right] \qquad (4.1)$$

where,

$$a_0 = \frac{1}{T} \int_0^{T_\emptyset} x_a(t)\ dt, \quad m = 0 \qquad a_m = \frac{2}{T} \int_0^{T_\emptyset} x_a(t).\cos\ (2\pi.f_m.t)\ dt \quad m = 1, 2, \ldots, M-1$$

and,

$$b_m = \frac{2}{T} \int_0^{T_\varnothing} x_a(t).\sin\,(2\pi.f_m.t)\ dt \qquad m = 0,\ 1,\ \ldots,\ M-1$$

M denotes the number of frequency components (including d.c.) and f_m relates to the frequency of each component.

Let samples of $x_a(t)$ be taken at pseudo-random intervals for digital processing. The magnitude of a sample at any instance, t_k is given by, $x_a(t_k)$ where k = 0 to N - 1 and N is the number of consecutive samples. Let the time domain sampling function, $u_0(t)$, be defined as,

$$u_0(t) = \sum_{k=-\infty}^{\infty} \delta(t - t_k) \qquad (4.2)$$

where $\delta(t)$ represents the impulse function. $u_0(t)$ is represented graphically in figure 4.1.

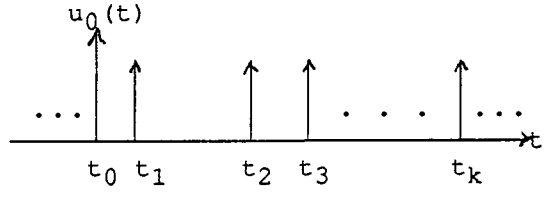


Figure 4.1. A series of impulses, each corresponding to a sampling instance.

The truncation due to taking a finite number of N samples of $x_a(t)$ in the time domain results in rippling in the frequency domain. The sampled, truncated function can be written as,

$$x_a(t).u_0(t).h(t) = x_a(t).\sum_{k=0}^{N-1} \delta(t - t_k) = \sum_{k=0}^{N-1} x_a(t_k).\delta(t - t_k) \qquad (4.3)$$

where h(t) is a rectangular window function of width $t_{N-1} + \tau_{min} = T_w$, given by,

$$h(t) = \begin{cases} 1 & \text{for} \quad -\tau_{min}/2 < t < t_{N-1} + \tau_{min}/2 \\ 1/2 & \text{for} \quad t = -\tau_{min}/2,\ t = t_{N-1} + \tau_{min}/2 \\ 0 & \text{otherwise} \end{cases} \qquad (4.4)$$

τ_{min} is the minimum possible difference between one sampling instance and the next, and t_{N-1} is the time the final sample is taken.

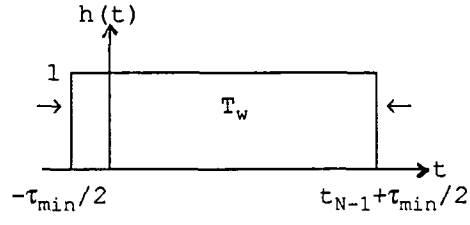


Figure 4.2. Rectangular window function.

There are N instantaneous samples of the periodic signal $x_a(t)$ in the interval T_w of the rectangular window function. These N samples are assumed to represent at least one period, T_ϕ of the signal $x_a(t)$; thus, it is assumed that $r.T_\phi = T_w$ where r is a positive integer ($r > 0$.) However, T_ϕ is fixed for a particular input signal, but $T_w = t_{N-1} + \tau_{min}$ depends on the time the final sample was taken, t_{N-1} which in turn depends on a pseudo-random quantity. Therefore, any practical system can only ensure that T_w is approximately equal to $r.T_\phi$. The consequences of this will be considered later.

For a Fourier transform of the samples to be made, the periodic function $x_a(t)$ must be modelled by the samples in the interval T_w. In order to form a periodic function $x_r(t)$ which consists of N samples repeated in the time domain at intervals of T_w, it is

necessary to use convolution. Repetition in the time domain, is equivalent to convolving the sampled, truncated waveform of equation (4.3) with the time function,

$$u_1(t) = T_w . \sum_{r=-\infty}^{\infty} \delta(t - rT_w)$$

The desired relationship is $x_r(t) = [x_a(t).u_0(t).h(t)] * u_1(t)$; hence,

$$x_r(t) = \left[\sum_{k=0}^{N-1} x_a(t_k).\delta(t - t_k) \right] * \left[T_w . \sum_{r=-\infty}^{\infty} \delta(t - rT_w) \right]$$

giving,

$$x_r(t) = T_w . \sum_{r=-\infty}^{\infty} \left[\sum_{k=0}^{N-1} x_a(t_k).\delta(t - t_k - rT_w) \right] \qquad (4.5)$$

This convolution result is a periodic function with period T_w that consists of N samples of the signal $x_a(t)$ where $T_w = t_{N-1} + \tau_{min}$ must equal the periodicity of $x_a(t)$, T_ϕ to prevent discontinuities. That is to say, $x_r(t)$ is an infinitely long sequence of the samples of $x_a(t)$ within the rectangular window h(t), with period T_w. However, if the period of $x_r(t)$, T_w is only approximately equal to an integer multiple of the period of the original signal $x_a(t)$, T_ϕ as stated earlier, then $x_r(t)$ will model a signal with discontinuities at intervals of T_w and the Fourier transform of $x_r(t)$ will yield only an approximation to the continuous Fourier transform of $x_a(t)$.

The requirement for repetition in the time domain (achieved by convolution with $u_1(t)$) affects the spectrum. In the frequency domain, this convolution is equivalent to multiplying the continuous spectrum of $x_a(t).u_0(t).h(t)$ by the function,

$$U_1(f) = \sum_{n=-\infty}^{\infty} \delta(f - n/T_w)$$

This is analogous to sampling in the frequency domain and so the approximate Fourier coefficients describing the signal can only be evaluated at discrete frequencies with a minimum separation of $1/T_w$. The Fourier transform of the periodic function $x_r(t)$ (with period T_w) is given by the sequence of impulses,

$$X_r'(f) = \sum_{n=-\infty}^{\infty} \alpha_{fn}.\delta(f - f_n) \quad \text{where} \quad \alpha_{fn} = \frac{1}{T_w}.\int_{-\tau_{min}/2}^{t_{N-1}+\tau_{min}/2} x_r(t).e^{-j2\pi f_n t}\,dt \qquad (4.6)$$

where the Fourier coefficients are calculated at regular frequency intervals such that the frequencies $f_n = n/T_w$, n = ... 1, 2,.... Substituting for $x_r(t)$ from equation (4.5) gives,

$$\alpha_{fn} = \frac{1}{T_w}.\int_{-\tau_{min}/2}^{t_{N-1}+\tau_{min}/2} T_w.\sum_{r=-\infty}^{\infty}\left[\sum_{k=0}^{N-1} x_a(t_k).\delta(t - t_k - rT_w)\right].e^{-j2\pi f_n t}\,dt$$

Note that the integral is only over one period (since $T_w = t_{N-1} + \tau_{min}$;) hence,

$$\alpha_{fn} = \int_{-\tau_{min}/2}^{t_{N-1}+\tau_{min}/2} \sum_{k=0}^{N-1} x_a(t_k).\delta(t - t_k).e^{-j2\pi f_n t}\,dt$$

$$= \sum_{k=0}^{N-1} x_a(t_k).\int_{-\tau_{min}/2}^{t_{N-1}+\tau_{min}/2} \delta(t - t_k).e^{-j2\pi f_n t}\,dt$$

$$= \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi f_n t_k} \qquad (4.7)$$

Therefore, by substituting (4.7) into (4.6) the Fourier transform of the function $x_r(t)$ is,

$$X_r'(f) = \sum_{n=-\infty}^{\infty} \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi f_n t_k}.\delta(f - f_n) \qquad (4.8)$$

Equation (4.8) is periodic and can be expressed equivalently as the desired pseudo-random discrete Fourier transform of equation (4.9) which describes one period of the function $X_r'(f)$.

$$X_r(f_n) = \hat{a}_{fn} - j\hat{b}_{fn} = \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi f_n t_k} \qquad (4.9)$$

where f_n is the particular frequency for which the estimated Fourier coefficients wish to be known. Thus, equation (4.9) gives the spectral property of a finite sequence of pseudo-random samples from the magnitude of the samples and the instances at which they are taken.

The pseudo-random DFT of equation (4.9) reduces to the conventional discrete Fourier transform when samples are taken at regular intervals. In the case of uniformly spaced sampling intervals, each sampling instance $\{t_k | k = 1 \text{ to } N-1\}$ can be represented as $t_k = k.T_s$ where T_s is the regular time interval between one sample and the next. The duration of the rectangular window, $T_w = t_{N-1} + \tau_{min} = (N-1).T_s + T_s = N.T_s$. Thus, the frequencies for which the estimated Fourier coefficients wish to be known, $f_n = n/T_w = n/(N.T_s)$. Substituting $f_n = n/(N.T_s)$ and $t_k = k.T_s$ into equation (4.9) reduces it to the familiar conventional DFT,

$$X_r(n/NT_s) = \sum_{k=0}^{N-1} x_a(k.T_s).e^{-j2\pi nk/N} \qquad (4.10)$$

The properties of the new discrete transform developed will depend upon the characteristics of the sampling instances. Before the periodicity and symmetrical properties of the transform are investigated, a way must be found to describe the production of pseudo-random sampling instances.

4.3 Generators of Pseudo-random Sampling Instances.

Two random sampling schemes have been extensively studied in **[14, Masry]** for their theoretical ability to form alias-free spectra. These are "additive random sampling", where the sampling instances are given by,

$$t_0 = 0, \quad t_k = t_{k-1} + \tau, \quad k = 1, 2, \ldots, N-1 \tag{4.11}$$

with τ as an independent positive random variable; and "periodic sampling with jitter", where the sampling instances are given by,

$$t_0 = 0, \quad t_k = k.T_s + \tau, \quad k = 1, 2, \ldots, N-1 \tag{4.12}$$

with τ as an independent random variable with zero mean over $[-T_s/2, T_s/2]$.

Consider the general case in which the signal input is sampled by some irregular pattern. In a practical system, the pattern must be limited such that the signal is sampled with a maximum and a minimum frequency and at a finite number of intermediate sampling frequencies. Thus, the random variable τ in the two sampling schemes must be considered as a pseudo-random discrete quantity which can be generated by some known function, R ( ) .

Let P represent the total number of possible sampling frequencies, which may be any arbitrary positive value.

Let the number produced by some pseudo-random number generator with a large sequence period take a value *between* zero and one with some controllable distribution (for example; uniform, Poisson or Gaussian,) and be represented by the function, R ( ). Let R ( ) never be equal to zero or one, but take values in between; ie. $0 < R( ) < 1$. One such pseudo-random number generator which produces a value *between* zero and one with an approximately uniform distribution, is described in **[15, Widrow & Stearns]**. Let the number produced by this generator be represented by the function, $R_u( )$. The function R ( ) is to be used in generating the irregularly spaced sampling instances. $R_u( )$ (a special case of R ( )) is the simplest pseudo-random function to implement and is used in the simulation program of appendix C.

4.3.1 Additive Pseudo-random Sampling.

Additive pseudo-random sampling instances are given by,

$$t_0 = 0, \quad t_k = t_{k-1} + \tau, \quad k = 1, 2, \ldots, N-1 \qquad (4.13)$$

where τ is an independent positive pseudo-random variable. τ will take one of P values and the value chosen will depend on the pseudo-random function R ( ).

Illustrative Example 4.1.

Let there be a total of three possible sampling frequencies; ie. P = 3, and assign the set of sampling frequencies, f_s as 12.0 Hz, 12.2 Hz and 12.4 Hz. Letting $\tau = 1/f_s$ means τ can take the values 1/12.0 s, 1/12.2 s and 1/12.4 s, in this case. If the independent function R ( ) returns a value less than or equal to 1/3 then τ is assigned the value 1/12.0 s; and if R ( ) returns a value greater than 1/3 but less than or equal to 2/3 then τ is assigned the value 1/12.2 s; otherwise τ is set to 1/12.4 s. If the pseudo-random function $R_u( )$ is used, τ will take one of these three values with approximately equal probability.

In general, τ takes one of a set of P deterministic values $\tau_1, \tau_2, ..., \tau_p$, and τ is assigned the value τ_χ where,

$$\chi = \lceil R() * P \rceil \text{ and } \lceil x \rceil \text{ denotes the integer ceiling of } x \qquad (4.14)$$

Consider the example 4.1 once again where $\tau_1 = 1/12.0$, $\tau_2 = 1/12.2$ and $\tau_3 = 1/12.4$. Note $t_0 = 0$. For each successive sampling instance, τ will take one of the three values τ_1, τ_2, or τ_3, thus,

$t_0 = 0$

$t_1 = 1/12.0$ or $1/12.2$ or $1/12.4$

$t_2 = 2/12.0,\ 1/12.0 + 1/12.2,\ 1/12.0 + 1/12.4,\ 2/12.2,$
$1/12.2 + 1/12.4$ or $2/12.4$

$t_3 = 3/12.0,\ 2/12.0 + 1/12.2,\ 2/12.0 + 1/12.4,\ 1/12.0 + 2/12.2,$
$1/12.0 + 1/12.2 + 1/12.4,\ 1/12.0 + 2/12.4,\ 3/12.2,$
$2/12.2 + 1/12.4,\ 1/12.2 + 2/12.4$ or $3/12.4$ ... and so on.

In general, given a sampler with P possible values that τ may take, the n^{th} sampling instance, t_{n-1} can take one of,

$$\xi_n^P = \frac{(n + P - 2)!}{(n - 1)!(P - 1)!} \qquad (4.15)$$

different values and the sampling instance t_k can be represented as,

$$t_k = u_1.\tau_1 + u_2.\tau_2 + \ldots + u_p.\tau_p \qquad (4.16)$$

where $\{u_i | i = 1 \text{ to } P\}$ are positive integers.

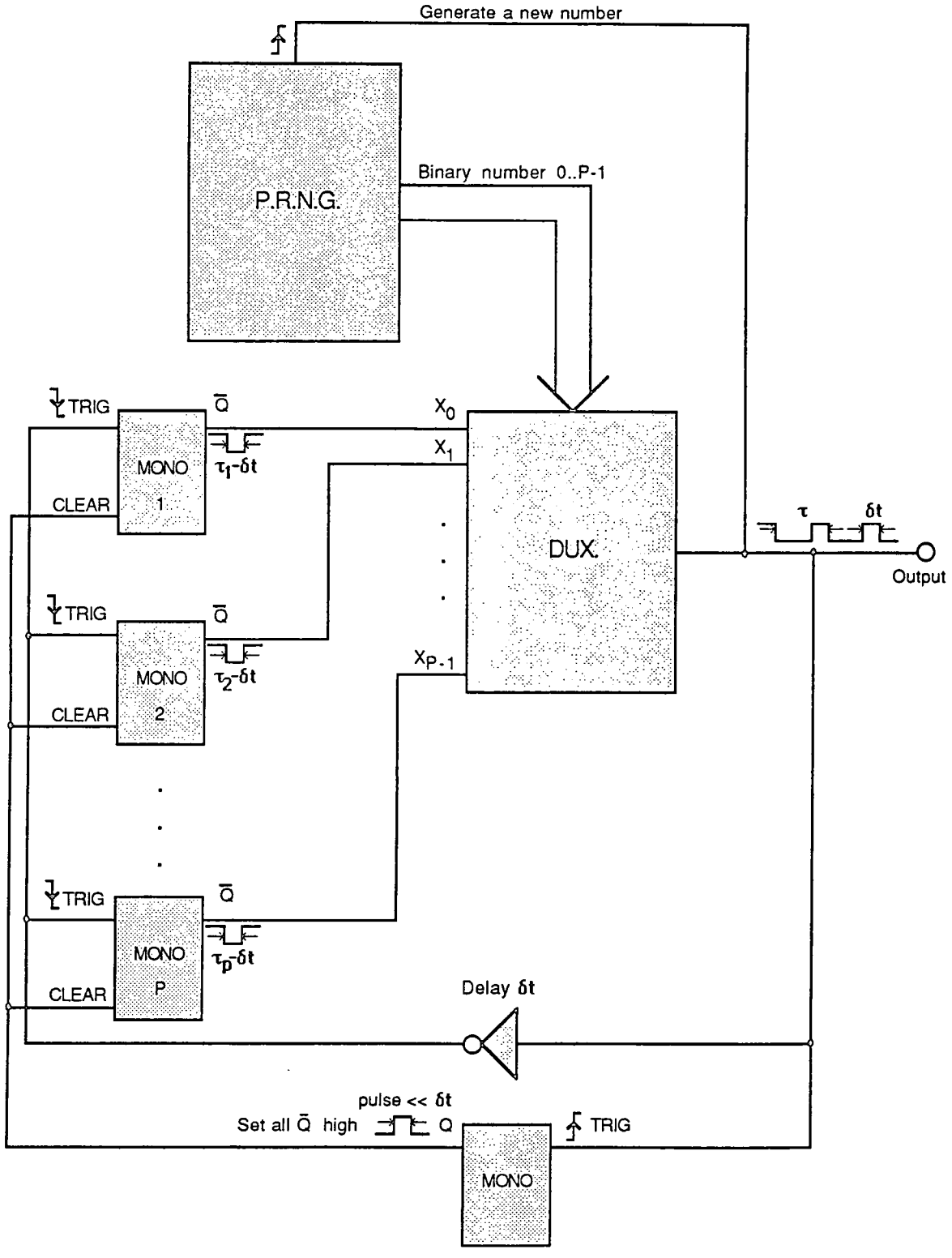


Figure 4.3. Proposed System to Produce Additive Pseudo-random Sampling Instances.

The minimum possible difference in sampling instances,

$$\tau_{min} = \text{the minimum value in the set } \{\tau_i | i = 1 \text{ to } P\}. \qquad (4.17)$$

In practice, it is proposed that a stream of sampling pulses, as described above, can be generated by the system shown in figure 4.3 which uses a series of monostables, each with a different pulse width and all capable of being forced to a stable state at any time. The P.R.N.G. module generates a pseudo-random integer, $\chi - 1$ in binary form in the region from 0 to P - 1, where χ is given by equation (4.14). This then acts as input to the demultiplexing module, DUX. which selects the corresponding monostable pulse as its output. On the rising edge of the output pulse, all the monostable outputs are set high, a new pseudo-random number is presented as input to the DUX. module, and after a short delay, the monostables are retriggered. In this way, a pulse of constant width is produced after varying delays and the system yields a train of pulses that can be used by an sample and hold circuit and that has sampling instances characterised by equation (4.16).

4.3.2 Periodic Sampling with Dither.

Periodic sampling instances with dither are given by,

$$t_0 = 0, \quad t_k = k.T_s + \tau, \quad k = 1, 2, \ldots, N-1 \qquad (4.18)$$

where τ is an independent pseudo-random variable with zero mean over $[-T_s/2, T_s/2]$. τ takes one of a set of P deterministic values $\tau_1, \tau_2, \ldots, \tau_p$, and the value chosen depends on the pseudo-random function R(), as with the previously described sampling scheme; ie. τ is assigned the value τ_χ where,

$$\chi = \lceil R() * P \rceil$$

Furthermore, the possible values that τ can take are fractional parts of $\pm T_s/2$. This ensures that $t_{k+1} > t_k$. τ is selected from the set,

$$\tau = \{\pm x_1.T_s, \pm x_2.T_s, \ldots, \pm x_z.T_s\} \quad (4.19)$$

where x_i are rational numbers such that,

$$0 \leq x_i < 1/2 \mid i = 1 \text{ to } z, \quad x_1 = 0 \text{ if } P \text{ is odd}, \quad \text{and} \quad z = \lceil P / 2 \rceil$$

The n^{th} sampling instance, t_{n-1} can take one of only P different values and the sampling instance t_k can be represented as,

$$t_k = T_s.(k \pm x_i) \quad (4.20)$$

The minimum possible difference in sampling instances,

$$\tau_{min} = \text{the minimum value in the set } \{T_s.(1 - 2x_i) \mid i = 1 \text{ to } P\}. \quad (4.21)$$

Illustrative Example 4.2.

A sampling scheme with a regular period, $T_s = 1/20.25$ s and five (P = 5) possible dithers $\tau = \{0, \pm 0.3T_s, \pm 0.4T_s\}$ will have sampling instances such that,

$t_0 = 0$

$t_1 = 1/20.25$ or $(1 \pm 0.3)/20.25$ or $(1 \pm 0.4)/20.25$

$t_2 = 2/20.25$ or $(2 \pm 0.3)/20.25$ or $(2 \pm 0.4)/20.25$

$t_3 = 3/20.25$ or $(3 \pm 0.3)/20.25$ or $(3 \pm 0.4)/20.25$

... and so on.

4.4 Transform Period and Input Signal Bandwidth Limitations.

It is necessary to determine the periodicity and symmetry of $X_r'(f)$ to find the bandwidth limitations that must be imposed on the input signal $x_a(t)$ so as to prevent frequency domain aliasing, and hence allow error-free signal reconstruction from the samples taken. Let f_p be the periodicity of the function $X_r'(f)$. The value of f_p is required such that $X_r(f_r + f_p) = X_r(f_r)$. Letting $f_n = f_r + f_p$; equation (4.9) becomes,

$$X_r(f_r + f_p) = \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi.(f_r + f_p).t_k}$$

$$= \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi.f_r t_k}.e^{-j2\pi.f_p t_k} \qquad (4.22)$$

Therefore, $X_r(f_r + f_p) = X_r(f_r)$ if and only if $e^{-j2\pi.f_p t_k} = 1$ for k integer valued from 0 to N-1. That is to say, $X_r'(f)$ has a period f_p, where f_p is the least positive number greater than zero such that,

$$f_p.t_k \text{ is integer valued for all } t_k,\ k = 0,\ 1,\ \ldots,\ N-1 \qquad (4.23)$$

4.4.1 Periodicity for Uniform Sampling Scheme.

Consider the simple case of uniform sampling in which the difference between one sampling instance, t_k and the next sampling instance, t_{k+1} is constant; ie. τ takes only one possible value, τ_c when compared with additive pseudo-random sampling. From equation (4.16), $t_k = u.\tau_c$ where u is a positive integer. Thus, substituting into (4.23), $X_r'(f)$ has a period f_p where, f_p is the least positive number greater than zero such that, $f_p.u.\tau_c$ is integer valued. That is obviously when $f_p = 1/\tau_c$ as u is an integer. However, $1/\tau_c$ is the uniform sampling frequency f_s. That is to say, the transform $X_r'(f)$ has a

period equal to the uniform sampling frequency, as expected from the traditional Fourier transform properties.

4.4.2 Periodicity for Additive Pseudo-random Sampling Scheme.

In the more complex case of additive pseudo-random sampling, each sampling instance is given by the general equation (4.16). Substituting this into equation (4.23) gives the period f_p as the least positive number greater than zero such that,

$$f_p.[u_1.\tau_1 + u_2.\tau_2 + \ldots + u_p.\tau_p] = K \text{ is an integer.}$$

By definition, $\{u_i | i = 1 \text{ to } P\}$ are positive integers, and so K is an integer if,

$$\{f_p.\tau_i | i = 1 \text{ to } P\} \text{ are integer values.} \qquad (4.24)$$

Equation (4.24) must be solved for f_p, the transform period. This may be done by first representing each $\{\tau_i | i = 1 \text{ to } P\}$ as a rational number in the most optimum form; ie. with use of the minimum possible denominator. Let each value that τ can take,

$$\{\tau_i = a_i / b_i | i = 1 \text{ to } P\} \text{ where } a_i \text{ and } b_i \text{ are integers.} \qquad (4.25)$$

If every member of $\{\tau_i | i = 1 \text{ to } P\}$ is multiplied by the lowest common multiple of the set of denominators $\{b_i | i = 1 \text{ to } P\}$, an integer will result. However, this lowest common multiple† will not be the smallest possible number that will produce an integer value if the greatest common divisor‡ of the set of numerators $\{a_i | i = 1 \text{ to } P\}$ is not equal to one. The smallest possible number that when multiplied by each and every

---

†The lowest common multiple (lcm) of two integers u and v, is the smallest positive integer that is a multiple of (ie., evenly divisible by) both u and v; the lcm of zero and zero is zero; and the lcm of one integer is that integer.

‡The greatest common divisor (gcd) of two positive integers m and n, is the largest positive integer which evenly divides both m and n.

member of $\{\tau_i | i = 1 \text{ to } P\}$ results in an integer; ie. f_p that satisfies equation (4.24) (and hence equation (4.23)), is given by,

$$f_p = \frac{\text{lowest common multiple } \{b_i | i = 1 \text{ to } P\}}{\text{greatest common divisor } \{a_i | i = 1 \text{ to } P\}}, \quad \text{for} \quad P > 1 \qquad (4.26)$$

Methods using Euclid's algorithm to determine the lowest common multiple and the greatest common divisor of a set of integers are described in **[16, Knuth]**.

For the example 4.1, the possible values of τ can be expressed in their optimum rational forms as $\tau_1 = 1/12$ s, $\tau_2 = 5/61$ s and $\tau_3 = 5/62$ s. Giving,

$$f_p = \frac{\text{lcm } (12,\ 61,\ 62)}{\text{gcd } (1,\ 5,\ 5)} = 22{,}692 \text{ Hz.}$$

4.4.3 Periodicity for Periodic Sampling Scheme with Dither.

For the periodic sampling scheme with dither, each sampling instance is given by equation (4.20). Substituting this into equation (4.23) gives the transform period f_p as the least positive number greater than zero such that,

$$f_p . T_s . (k \pm x_i) = K \text{ is an integer for } i = 1 \text{ to } z. \qquad (4.27)$$

Note k is an integer by definition. Let T_s and $\{T_s.x_i | i = 1 \text{ to } z\}$ be represented as rational numbers with the minimum possible denominator such that,

$$T_s = a_0 / b_0 \quad \text{and} \quad \{T_s . x_i = a_i / b_i | i = 1 \text{ to } z\} \qquad (4.28)$$

where all a_i and b_i are integers. In the same manner f_p was derived to satisfy equation (4.24), the hypothesis of equation (4.27) is satisfied by,

$$f_p = \frac{\text{lowest common multiple } \{b_i | i = 0 \text{ to } z\}}{\text{greatest common divisor } \{a_i | i = 0 \text{ to } z\}}, \quad \text{for} \quad P > 1 \qquad (4.29)$$

For the example 4.2, the regular period, $T_s = 1/20.25$ s and the three values of $T_s.x_i = \{0, 0.3T_s, 0.4T_s\}$ can be expressed in their optimum rational forms as $T_s = 4/81$, $T_s.x_0 = 0$, $T_s.x_1 = 8/405$, and $T_s.x_2 = 2/135$. Giving the transform period as,

$$f_p = \frac{\text{lcm } (81,\ 405,\ 135)}{\text{gcd } (4,\ 8,\ 2)} = 202.5 \text{ Hz.}$$

4.4.4 System Bandwidth.

Now consider,

$$X_r(f_p - f_r) = \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi.(f_p - f_r).t_k}$$

$$= \sum_{k=0}^{N-1} x_a(t_k).e^{-j2\pi.f_p t_k}.e^{+j2\pi.f_r t_k}$$

$$= \sum_{k=0}^{N-1} x_a(t_k).e^{+j2\pi.f_r t_k}$$

$$= X_r^*(f_r) \qquad (4.30)$$

since $e^{-j2\pi.f_p t_k} = 1$ by definition of f_p from equation (4.23). Therefore, the amplitude of the transform, $|X_r(f_n)|$ is even-symmetrical about $f_p/2$ and its argument, $\angle [X_r(f_n)]$ is odd-symmetrical about $f_p/2$. That is, the transform consists of complex conjugates over one period, and so only half the information in one cycle is of interest. Therefore, the input signal must be band-limited to half the period of the transform in order to prevent frequency domain aliasing; ie. in the case of uniform sampling, the Nyquist criterion

must be satisfied. However, for pseudo-random sampling (additive or periodic with dither) the transform period is noticeably high. In the example 4.1, the signal needs only to be band-limited to 11,346 Hz although it is being sampled at frequencies of 12.0 Hz, 12.2 Hz, and 12.4 Hz. Similarly, in the example 4.2 where the maximum sampling frequency is only 67.5 Hz, the input signal only needs to be band-limited to 101.25 Hz. It appears that the Nyquist criterion need not be satisfied when using pseudo-random sampling! However, it has not yet been established as to whether or not the spectrum produced by the pseudo-random DFT is 'alias-free' for irregularly sampled signals limited within this enlarged bandwidth.

In general, for any known pseudo-random sampling generator, it is possible to determine the periodicity, f_p of the transform function $X_r'(f)$ by the method described above, and hence the bandwidth limitation, B_{pseudo} that must be imposed on the input signal; that is,

$$B_{pseudo} = \frac{1}{2} \cdot \frac{\text{lowest common multiple } \{b_i | \text{ for all } i\}}{\text{greatest common divisor } \{a_i | \text{ for all } i\}} \qquad (4.31)$$

where all a_i and b_i are defined for the additive pseudo-random sampling scheme and for the periodic sampling scheme with dither by equations (4.25) and (4.28) respectively. Note that the bandwidth B_{pseudo} is independent of the total number of possible sampling frequencies, P and only depends upon the possible changes between one sampling instance and the next for additive pseudo-random sampling, and upon the possible dithers and regular sampling period for periodic sampling with dither.

Illustrative Example 4.3.

An illustration of this is given in figure 4.4, which shows the output of the transform due to sampling a signal containing its fundamental harmonic at 17.0 Hz and no

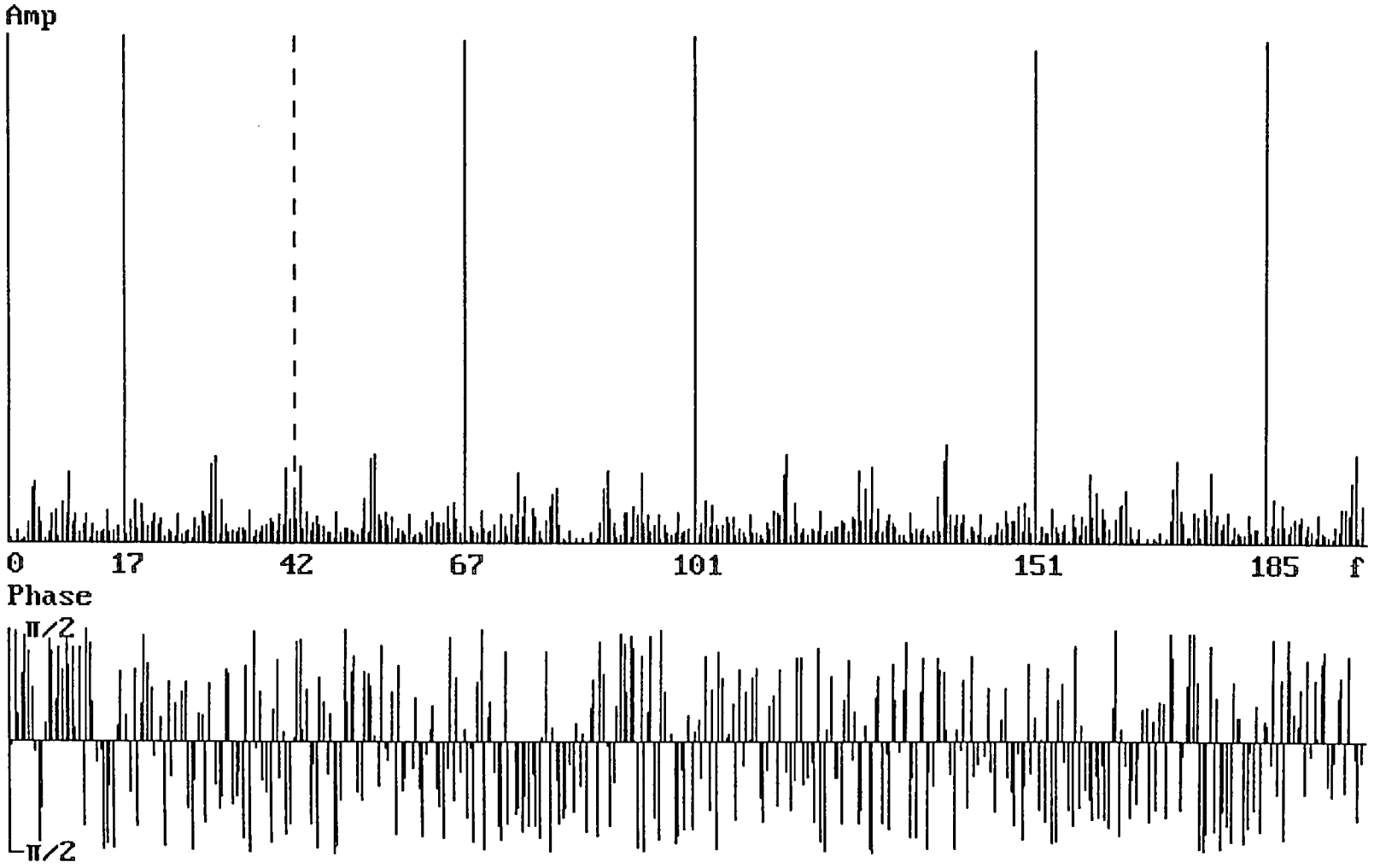


Figure 4.4. Pseudo-random sampling at 12.0 Hz & 14.0 Hz, 500 samples taken
Single element input at 17.0 Hz, amplitude 1.0, phase 0°

overtones. In this case, the additive pseudo-random sampling process involved five hundred samples being taken at intervals of 1/12.0 s or 1/14.0 s; hence, $f_p = 84.0$ Hz from equation (4.29). It can clearly be seen that the input signal needs to be band-limited to $f_p/2 = 42.0$ Hz.

4.5 Inverse Pseudo-random DFT.

A transform has been defined which takes pseudo-random samples of a signal $x_a(t)$ as its input data and produces an output $X_r'(f)$. An inverse transform is required to find the N samples $x_a(t_k)$, $t_k = t_0, t_1, ..., t_{N-1}$ from the data $X_r'(f)$.

If the sampling instances t_k, $k = 0, 1, ..., N-1$ are spaced uniformly by intervals T_s, then the inverse transform may be represented as **[5, Brigham]**,

$$x_a'(t_k) = \frac{1}{N} \sum_{n=0}^{N-1} X_r(f_n) . e^{+j2\pi . f_n t_k}$$

where, $f_n = \dfrac{n}{N . T_s}$ and $t_k = k . T_s$ (see also, equation (4.10)) (4.32)

Note that the derivation of this inverse transform (for uniformly spaced samples) relies on the orthogonality relationship **[5, Brigham]**,

$$\sum_{n=0}^{N-1} e^{-j2\pi . f_n t_\lambda} . e^{+j2\pi . f_n t_\kappa} = \begin{cases} N & \text{for} \quad \lambda = \kappa \\ 0 & \text{otherwise} \end{cases} \qquad (4.33)$$

which does not hold for irregularly spaced samples. However, the two pseudo-random sampling schemes (defined in section 4.3) may be viewed as uniform sampling with a large number of carefully chosen missing sampling instances. Define the sequence,

$$x(\lambda) = \begin{cases} x_a(t_k) & \text{for} \quad \lambda/f_p \in \{t_k | k = 0 \text{ to } N-1\} \\ 0 & \text{otherwise} \end{cases} \quad \text{for} \quad \lambda = 0 \text{ to } Q-1 \tag{4.34}$$

where $Q = (t_{N-1} + \tau_{min}).f_p$ is the minimum sequence length required to accommodate every sample instance in one period of $x_a(t)$. Remember that in the theoretical development of the pseudo-random DFT in section 4.2, the periodicity of $x_a(t)$ was assumed to equal the duration, $T_w = t_{N-1} + \tau_{min}$ of the rectangular window function h(t). τ_{min} is as defined by equations (4.17) and (4.21) for the additive pseudo-random and periodic with dither sampling schemes respectively. Clearly Q is an integer by definition of f_p in equation (4.23).

Equation (4.9) becomes,

$$X_r(f_n) = \sum_{\lambda=0}^{Q-1} x(\lambda).e^{-j2\pi.f_n.\lambda/f_p} \tag{4.35}$$

Let $f_n = n.\delta f$ where δf is the DFT frequency bin spacing, with $n = 0, 1, ..., f_p/\delta f - 1$. Thus,

$$X_r(n.\delta f) = \sum_{\lambda=0}^{Q-1} x(\lambda).e^{-j2\pi.n.\delta f.\lambda/f_p} \tag{4.36}$$

Consider the hypothesis that the inverse pseudo-random DFT is of the same form as equation (4.32); ie. that,

$$x(\kappa) = \frac{\delta f}{f_p} \sum_{n=0}^{f_p/\delta f - 1} X_r(n.\delta f).e^{+j2\pi.n.\kappa.\delta f/f_p} \tag{4.37}$$

for $\kappa = 0, 1, ..., Q-1$. Substituting equation (4.36) into equation (4.37) gives,

$$x(\kappa) = \frac{\delta f}{f_p} \sum_{n=0}^{f_p/\delta f - 1} \sum_{\lambda=0}^{Q-1} x(\lambda) . e^{-j2\pi . n . \lambda . \delta f / f_p} . e^{+j2\pi . n . \kappa . \delta f / f_p}$$

A swap of the summations is permissible only if $\delta f = 1/(t_{N-1} + \tau_{min})$. Thus,

$$x(\kappa) = \sum_{\lambda=0}^{Q-1} x(\lambda) \frac{\delta f}{f_p} \left[\sum_{n=0}^{Q-1} e^{-j2\pi . n . \lambda . \delta f / f_p} . e^{+j2\pi . n . \kappa . \delta f / f_p} \right] \quad (4.38)$$

Consider the section of this expression in the square brackets which is equivalent to,

$$\sum_{n=0}^{Q-1} e^{-j2\pi . n . \delta f / f_p . (\lambda - \kappa)} = V()$$

and let,

$$v = e^{+j\Omega(\kappa - \lambda)} \quad \text{where} \quad \Omega = 2\pi . \delta f / f_p \quad (4.39)$$

to give†,

$$V() = \sum_{n=0}^{Q-1} v^n = \frac{1 - v^Q}{1 - v} \quad \text{for} \quad v \neq 1 \quad (4.40)$$

From equation (4.39), $v = 1$ for $\lambda = \kappa$ and $v^Q = 1$ for $\lambda \neq \kappa$ as λ, κ and $\delta f/f_p$ are all integers by definition; so, substituting into (4.40) gives,

---

†From the mathematical principles of sums and products, the basic formula for the sum of a geometric progression is given by,

$$\sum_{0 \leq j \leq n} ax^j = a . \left[\frac{1 - x^{n+1}}{1 - x} \right]$$

assuming that $x \neq 1$ and $n \geq 0$. **[16, Knuth]**.

$$V() = \begin{cases} Q & \text{for} \quad \lambda = \kappa \\ 0 & \text{otherwise} \end{cases} \qquad (4.41)$$

Thus equation (4.38) becomes its identity, so the hypothesis of equation (4.37) must be valid and the inverse pseudo-random discrete Fourier transform is given by,

$$x_a'(t_k) = \frac{1}{Q} \sum_{n=0}^{Q-1} X_r(n.\delta f).e^{+j2\pi.t_k.n.\delta f} \quad \text{for} \quad t_k = t_0, t_1, \ldots, t_{N-1} \qquad (4.42)$$

where $Q = (t_{N-1} + \tau_{min}).f_p$ and $\delta f = 1/(t_{N-1} + \tau_{min})$.

The discrete inversion formula (4.42) exhibits periodicity defined by the N samples of $x_a'(t)$ in a manner similar to the discrete transform; such that,

$$x_a'(t_k) = x_a'(t_k + q.[t_{N-1} + \tau_{min}]) \quad \text{for} \quad q = 0, \pm 1, \pm 2, \ldots$$

Examination of the formula in (4.42) also reveals that to reconstruct the N sample values of $x_a(t)$ at $t = t_0, t_1, \ldots, t_{N-1}$ from $X_r(f_n)$ requires an excessive $Q = (t_{N-1} + \tau_{min}).f_p$ points in the frequency domain to be calculated. Thus, an estimated wideband spectral analysis of the input signal may be made rapidly by taking N samples at pseudo-random intervals and performing the transform described by equation (4.9), but the reconstruction of just N samples of the signal $x_a(t)$ at specific instances from this spectrum, although possible, involves vast time consuming evaluations.

The workload involved can be dramatically reduced by noting that in practice, the input $x_a(t)$ is real and so the values of $x_a'(t_k)$ for k = 0, 1, ..., N-1 given by the inverse transform must also be real. Remember that when $x_a(t)$ is real, equation (4.30) holds; ie.

$$X_r(f_p - f_r) = X_r^*(f_r)$$

thus,

$$X_r(Q.\delta f - r.\delta f) = X_r^*(r.\delta f)$$

and similarly,

$$X_r(r.\delta f) = X_r^*(Q.\delta f - r.\delta f)$$

The formula of (4.42) can therefore be reduced so that nearly only half the number of frequency bins need to be calculated. When Q is even,

$$\begin{aligned} x_a'(t_k) &= \frac{1}{Q}\left\{ \sum_{n=0}^{Q/2} X_r(n.\delta f).e^{+j2\pi.t_k.n.\delta f} \right. \\ &\quad \left. + \sum_{n=1}^{Q/2-1} X_r^*(n.\delta f).e^{+j2\pi.t_k.(Q-n).\delta f} \right\} \\ &= \frac{1}{Q}\left\{ X_r(0) + X_r(f_p/2).e^{+j\pi.t_k.f_p} \right. \\ &\quad \left. + 2\sum_{n=1}^{Q/2-1} \hat{a}(n\delta f)\cos(2\pi t_k n\delta f) + \hat{b}(n\delta f)\sin(2\pi t_k n\delta f) \right\} \end{aligned} \tag{4.43}$$

and when Q is odd,

$$\begin{aligned} x_a'(t_k) &= \frac{1}{Q}\left\{ \sum_{n=0}^{(Q-1)/2} X_r(n.\delta f).e^{+j2\pi.t_k.n.\delta f} \right. \\ &\quad \left. + \sum_{n=1}^{(Q-1)/2} X_r^*(n.\delta f).e^{+j2\pi.t_k.(Q-n).\delta f} \right\} \\ &= \frac{1}{Q}\left\{ X_r(0) \right. \\ &\quad \left. + 2\sum_{n=1}^{(Q-1)/2} \hat{a}(n\delta f)\cos(2\pi t_k n\delta f) + \hat{b}(n\delta f)\sin(2\pi t_k n\delta f) \right\} \end{aligned} \tag{4.44}$$

where $X_r(n.\delta f) = \hat{a}(n\delta f) - j.\hat{b}(n\delta f)$.

The value of $x_a'(t_k)$ yielded by equation (4.43) or equation (4.44) must be real. This is so as $X_r(0)$, $X_r(f_p/2)$ and $e^{+j\pi.t_k.f_p}$ will have no imaginary part by definition.

Illustrative Example 4.4.

The pseudo-random DFT of (4.9) and its inverse as described by equations (4.43) and (4.44) have been simulated for verification. (The simulation program source code is in Appendix C.) Figures 4.5 and 4.6 show the errors associated with the signal amplitude as determined by the inversion formulae at each sample instance in comparison with the actual signal amplitude. The input signal contained two harmonics; one at 7 Hz of amplitude 1 V, and another at 22 Hz of amplitude 2 V at 90° out of phase with the first harmonic. Figure 4.5 shows the error when the signal is sampled at 400 points uniformly at 84 Hz (system bandwidth 42 Hz.)

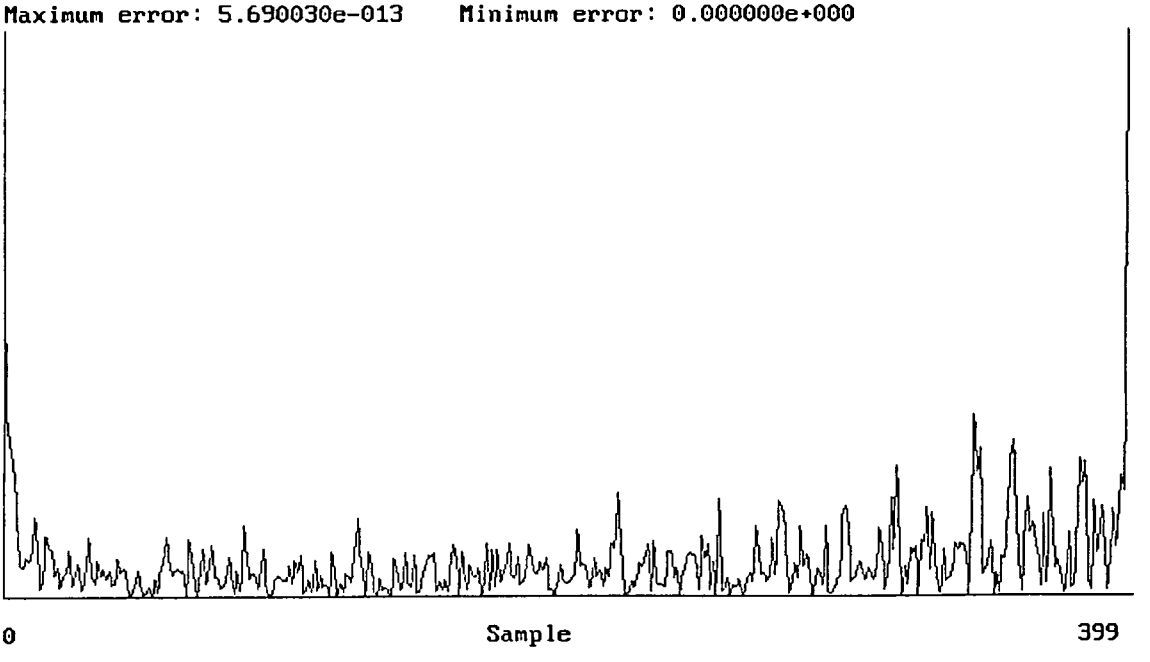


Figure 4.5. Example of amplitude errors from use of the inverse transform after sampling a signal uniformly.

The errors that result are relatively small, occur because of computational inaccuracies when using floating-point arithmetic, and compare favourably with the errors that are

produced when using the NAG (Numerical Algorithms Group) library routines for an analogous simulation. The NAG simulation program source code is listed in appendix D. When executed, the list of errors produced showed that the errors generated by taking a DFT of a sequence of samples and then performing an inverse DFT, are of similar magnitude to those shown in figure 4.3, with the error in the first and final samples considerably greater than the others.

Figure 4.6 shows the error when the signal is sampled at 400 points using the additive pseudo-random sampling scheme with τ_1 = 1/12.0 s and τ_2 = 1/14.0 s (system bandwidth 42 Hz.) The errors are again relatively small, occur due to floating-point arithmetic inaccuracies, but are on average slightly greater than the errors of figure 4.5 because more computation is required in the derivation of the sample values. In this case, the window width (the duration over which the 400 samples are taken) is approximately 400/13.0 s, compared with the case when the uniform sampling scheme is used for which the window width is only 400/84.0 s. Thus the number of frequency bins evaluated, Q = window width.f_p, is much greater when using irregular sampling.

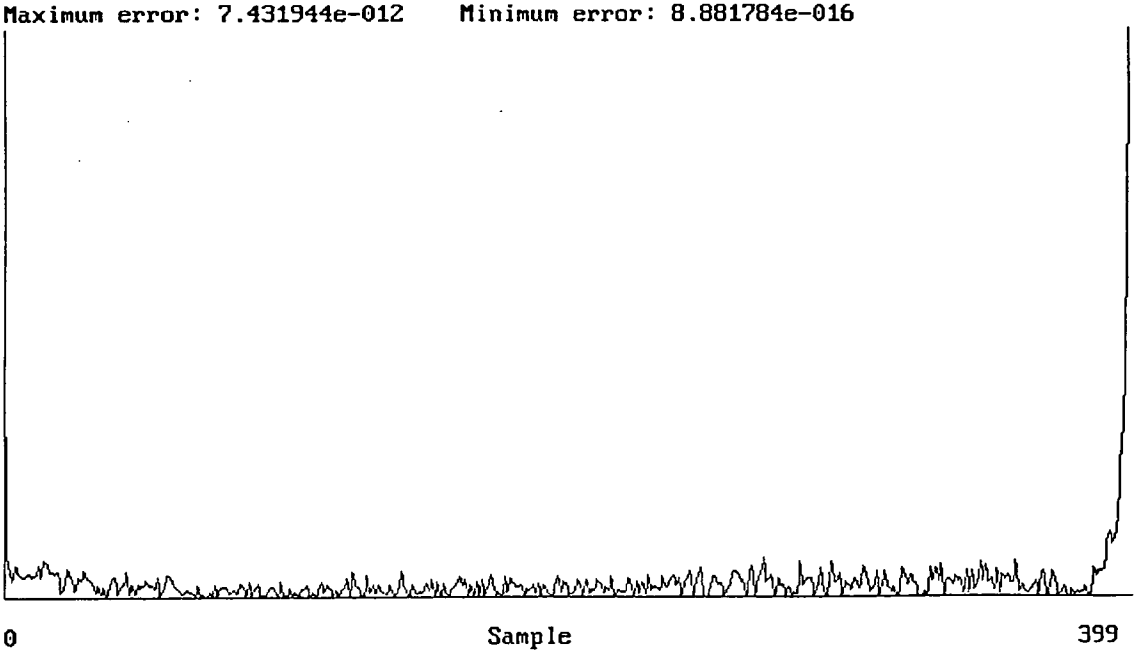


Figure 4.6. Example of amplitude errors from use of the inverse transform after sampling a signal irregularly.

The amplitude error is noticeably high for the first and final samples. This is a familiar characteristic of the conventional inverse discrete Fourier transform (evident when using the NAG library routines) and is also due to the use of floating-point arithmetic in the simulation. The analysis on this error is covered in some detail in a collection of papers compiled in [17, Liu].

It has been clearly shown and verified by simulation that the equation (4.9) is a transform with a well defined inverse given by equation (4.42). However, this does not mean that the original signal may be reconstructed from its irregular samples, only that instantaneous values of the signal can be determined from an estimated spectrum. A method is required to find out whether or not unambiguous signal reconstruction is possible when using an irregular sampling scheme. If such reconstruction is possible, relative to a particular sampling scheme, then that scheme will be alias-free.

4.6 Improving the Estimated Fourier Coefficients.

To show that it is possible to reconstruct the originally sampled signal from the N sample points, it is necessary to show that the coefficients a_m and b_m, used to describe the signal by equation (4.1) for each harmonic at a frequency f_m, can be determined without ambiguity from the N sample points.

The coefficients generated by the pseudo-random transform in equation (4.9) are estimates of the Fourier coefficients a_m and b_m, as is illustrated by figure 4.7.

Illustrative Example 4.5.

A signal containing one harmonic at 170 Hz of amplitude 1 V and another at 30 Hz of amplitude 2 V with a 90° phase difference was simulated as being sampled at 500 points using periodic sampling of 50 Hz ($T_s = 1/50$ s) with 5 possible dithers $\tau = \{0, \pm 0.1T_s, \pm 0.4T_s\}$ (giving a system bandwidth of 250 Hz). The spectrum of figure 4.7

shows the amplitude and phase of possible signal elements at frequencies spaced by 0.5 Hz.

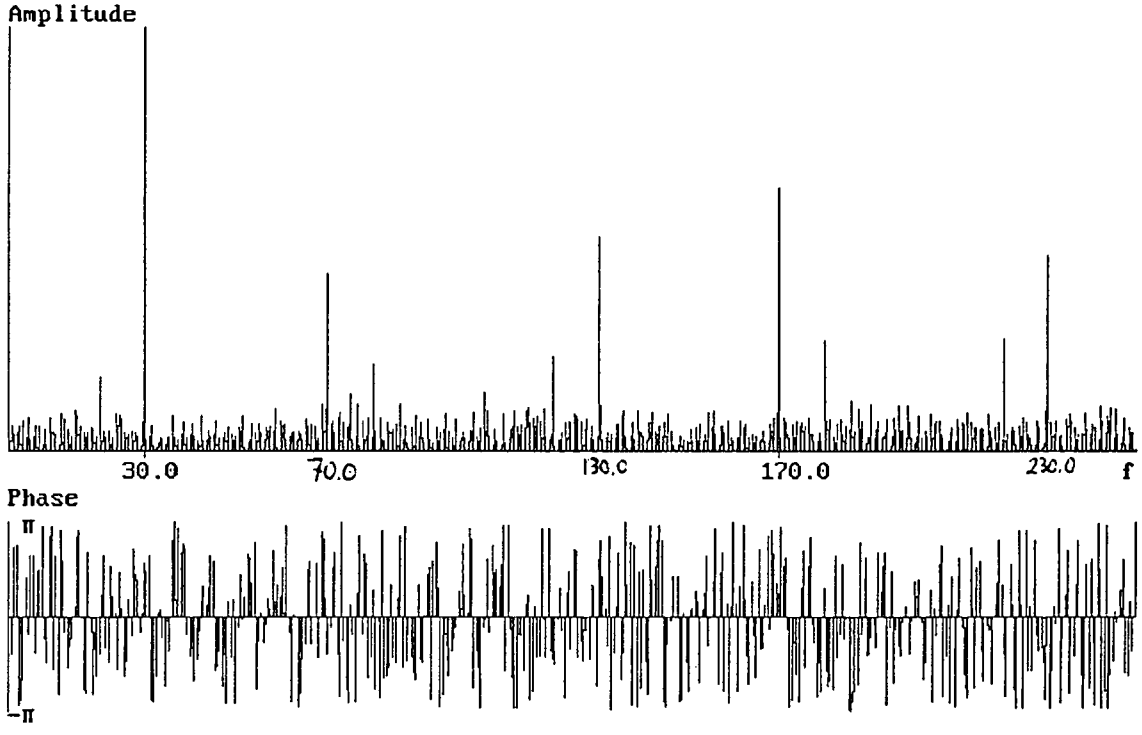


Figure 4.7. An illustration of the large amount of noise present in the spectrum formed from the pseudo-random Fourier transform of equation (4.9).

Examination of figure 4.7 reveals the possibility of at least three erroneous signal elements at frequencies of 70 Hz, 130 Hz and 230 Hz in addition to the true components at 30 Hz and 170 Hz.

It is necessary to devise an operation to eliminate this noise and thus determine the exact Fourier coefficients without ambiguity.

Note that $x_a(t_k)$ is given by (4.1). Substituting into (4.9) gives,

$$X_r(f_n) = \sum_{k=0}^{N-1} \sum_{m=0}^{M-1} \left[a_{fm}\cos(2\pi f_m t_k) + b_{fm}\sin(2\pi f_m t_k) \right] . e^{-j2\pi f_n t_k} \qquad (4.45)$$

Rearranging this for its real and imaginary parts gives,

$$\hat{a}_{fn} = \sum_{m=0}^{M-1} \sum_{k=0}^{N-1} \cos(2\pi f_n t_k) . \left[a_{fm} . \cos(2\pi f_m t_k) + b_{fm} . \sin(2\pi f_m t_k) \right] \tag{4.46}$$

and,

$$\hat{b}_{fn} = \sum_{m=0}^{M-1} \sum_{k=0}^{N-1} \sin(2\pi f_n t_k) . \left[a_{fm} . \cos(2\pi f_m t_k) + b_{fm} . \sin(2\pi f_m t_k) \right] \tag{4.47}$$

Equations (4.46) and (4.47) can be expressed in matrix form, as suggested by **[8, Bilinsky, Vystavkin & Mikelson]**; ie.,

$$\mathbf{B} = \mathbf{A}.\mathbf{C} \quad \text{or} \quad \mathbf{C} = \mathbf{A}^{-1}.\mathbf{B} \tag{4.48}$$

where the vectors,

$$\mathbf{C} = \begin{bmatrix} a_{f0} \\ a_{f1} \\ b_{f1} \\ \cdot \\ \cdot \\ \cdot \\ a_{f\mu} \\ b_{f\mu} \end{bmatrix} - \text{original coefficients sought} \qquad \mathbf{B} = \begin{bmatrix} \hat{a}_{f0} \\ \hat{a}_{f1} \\ \hat{b}_{f1} \\ \cdot \\ \cdot \\ \cdot \\ \hat{a}_{f\mu} \\ \hat{b}_{f\mu} \end{bmatrix} - \text{coefficients calculated from (4.9)}$$

and the system transfer matrix,

$$\mathbf{A} = \begin{bmatrix} \beta_{s00} & \alpha_{c10} & \beta_{s10} & \alpha_{c20} & \beta_{s20} & \cdots & \alpha_{cm0} & \beta_{sm0} \\ \alpha_{s01} & \alpha_{c11} & \alpha_{s11} & \alpha_{c21} & \alpha_{s21} & \cdots & \alpha_{cm1} & \alpha_{sm1} \\ \beta_{c01} & \beta_{c11} & \beta_{s11} & \beta_{c21} & \beta_{s21} & \cdots & \beta_{cm1} & \beta_{sm1} \\ \alpha_{s02} & \alpha_{c12} & \alpha_{s12} & \alpha_{c22} & \alpha_{s22} & \cdots & \alpha_{cm2} & \alpha_{sm2} \\ \beta_{c02} & \beta_{c12} & \beta_{s12} & \beta_{c22} & \beta_{s22} & \cdots & \beta_{cm2} & \beta_{sm2} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{c0m} & \alpha_{c1m} & \alpha_{s1m} & \alpha_{c2m} & \alpha_{s2m} & \cdots & \alpha_{cmm} & \alpha_{smm} \\ \beta_{s0m} & \beta_{c1m} & \beta_{s1m} & \beta_{c2m} & \beta_{s2m} & \cdots & \beta_{cmm} & \beta_{smm} \end{bmatrix} \tag{4.49}$$

with,

$$\alpha_{cij} = \sum_{k=0}^{N-1} \cos(2\pi f_i t_k).\cos(2\pi f_j t_k)$$

$$\alpha_{sij} = \sum_{k=0}^{N-1} \sin(2\pi f_i t_k).\cos(2\pi f_j t_k)$$

$$\beta_{cij} = \sum_{k=0}^{N-1} \cos(2\pi f_i t_k).\sin(2\pi f_j t_k)$$

$$\beta_{sij} = \sum_{k=0}^{N-1} \sin(2\pi f_i t_k).\sin(2\pi f_j t_k)$$

The dimensions of the matrix **A**, μ by μ, are governed by the periodicity of the pseudo-random DFT, $f_p = 2.B_{pseudo}$ and the frequency increment that the coefficients are calculated, δf. The estimated coefficients are evaluated at frequencies $f_m = m.\delta f$ for $m = 0$ to $\lfloor B_{pseudo}/\delta f \rfloor$. If $f_m = 0$ (ie. $m = 0$) or $f_m = B_{pseudo}$ (ie. $m = B_{pseudo}/\delta f$ is integer valued) then from the definition of the coefficients $X_r(f_m)$, b_{fm} will be zero. It is therefore unnecessary to find the imaginary part of the coefficients for the first frequency bin (ie. $f_m = 0$) at any time, or for the highest frequency bin (ie. $f_m = B_{pseudo}$) when $\lfloor f_p/\delta f \rfloor$ is even. Thus,

$$\mu = 2.\lfloor B_{pseudo}/\delta f \rfloor \ \{ \ + 1 \text{ if } \lfloor f_p/\delta f \rfloor \text{ is odd } \} \qquad (4.50)$$

The exact Fourier coefficients, **C** may be calculated from the inverse system transfer matrix, $\mathbf{A^{-1}}$ and the estimated Fourier coefficients, **B**. Matrix **A** is independent of the input signal and so **A** and $\mathbf{A^{-1}}$ may be calculated prior to sampling for a known set of sampling instances. Noting that, $\alpha_{s0j} = \alpha_{cj0}$, $\beta_{c0j} = \beta_{sj0}$, and $\alpha_{sij} = \beta_{cji}$, makes it clear to see **A** is a symmetrical matrix. It follows from simple matrix theory that $\mathbf{A^{-1}}$ must also

be symmetrical. This fact can be used to reduce the computation required to evaluate the matrix and its inverse.

The classical approach for determining a matrix inversion, based on the use of Cramer's rule **[18, Pipes & Hovanessian]**, involves an excessive number of arithmetical operations, approximately in the order of n^5. Using an augmented matrix method, based on Gauss' elimination, the inverse matrix $\mathbf{A^{-1}}$ may be obtained with a reduction in computation to the order of n^3 **[18, Pipes & Hovanessian]**. The most efficient method (implemented in the simulation program of appendix C,) also of order n^3 but with a reduced constant of proportionality, is LU decomposition based on Crout's algorithm and is described in detail by **[19 Press, Flannery, Teukolsky & Vettering]**.

It is expected that by using the inverse of the matrix in equation (4.49), the spectrum of a signal that has been band-limited in accordance with equation (4.31) and sampled irregularly at a maximum rate which may be below the Nyquist sampling rate, can be evaluated within the band limits without ambiguity at a finite number of frequencies.

Illustrative Example 4.6.

Consider, once again (as in the example 4.5,) a signal containing one harmonic at 170 Hz of amplitude 1 V and another at 30 Hz of amplitude 2 V with a 90° phase difference. The signal is simulated as being sampled at 500 points by two pseudo-random sampling schemes, each with a system bandwidth of 250 Hz. Scheme 1: Additive pseudo-random sampling with τ_1 = 1/100.0 s and τ_2 = 1/125.0 s. Scheme 2: Periodic sampling at 50 Hz (T_s = 1/50 s) with 5 possible dithers τ = {0, $\pm 0.1T_s$, $\pm 0.4T_s$}. Figure 4.8 shows the estimated signal spectrum (evaluated from equation (4.9)) when the signal is sampled using scheme 1. Figure 4.9 clearly shows that the noise in this estimated spectrum is eliminated when its coefficients are multiplied by the inverse of the matrix in equation (4.49), as expected.

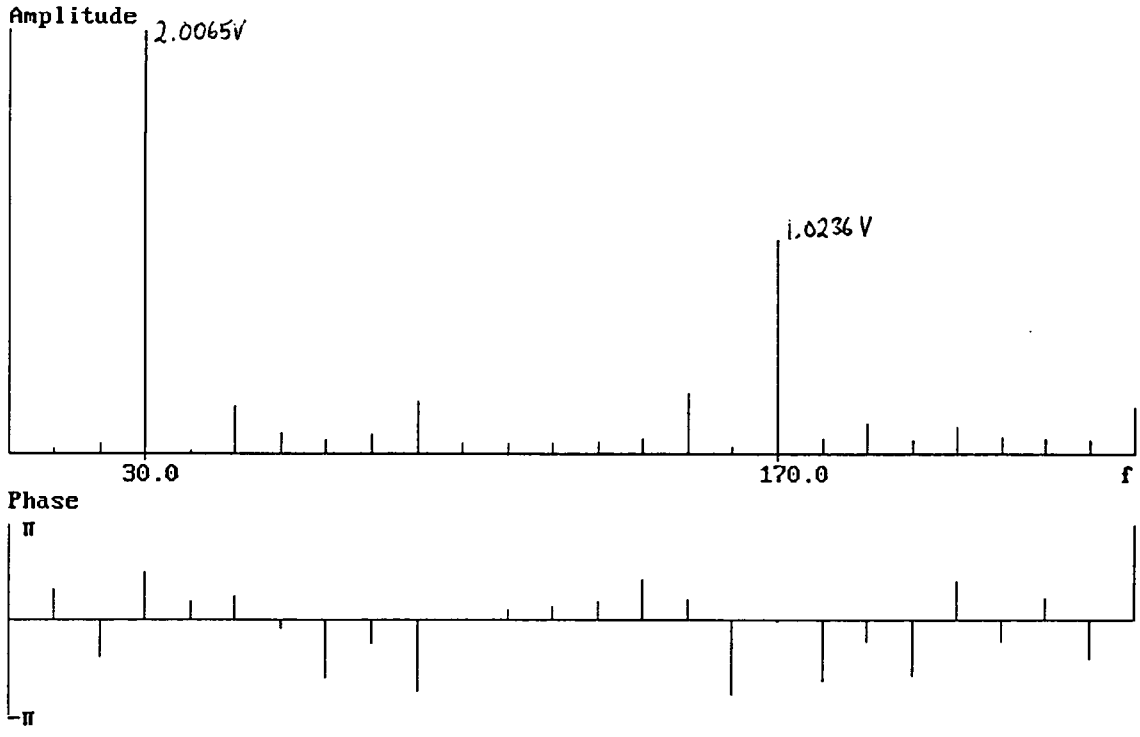


Figure 4.8. Estimated spectrum of example signal sampled by scheme 1.

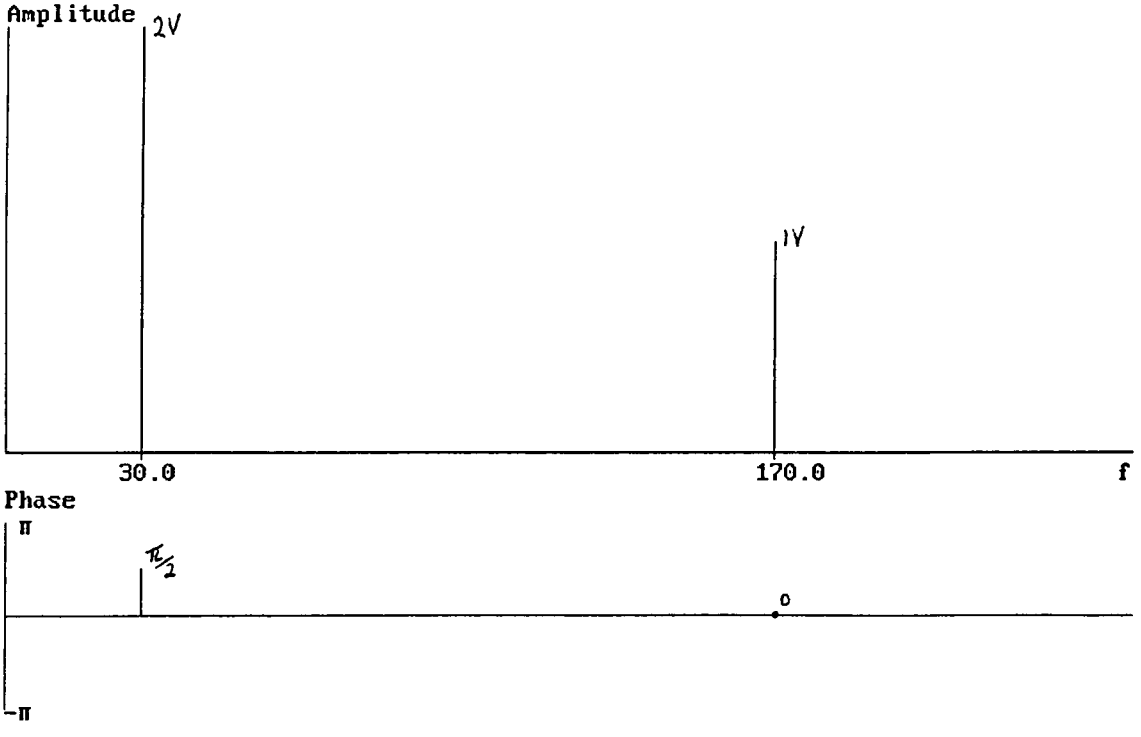


Figure 4.9. Improved spectrum of signal after additive pseudo-random sampling.

However, figures 4.10 and 4.11 show corresponding spectra when the signal is sampled using scheme 2. The noise is not eliminated and ambiguities remain.

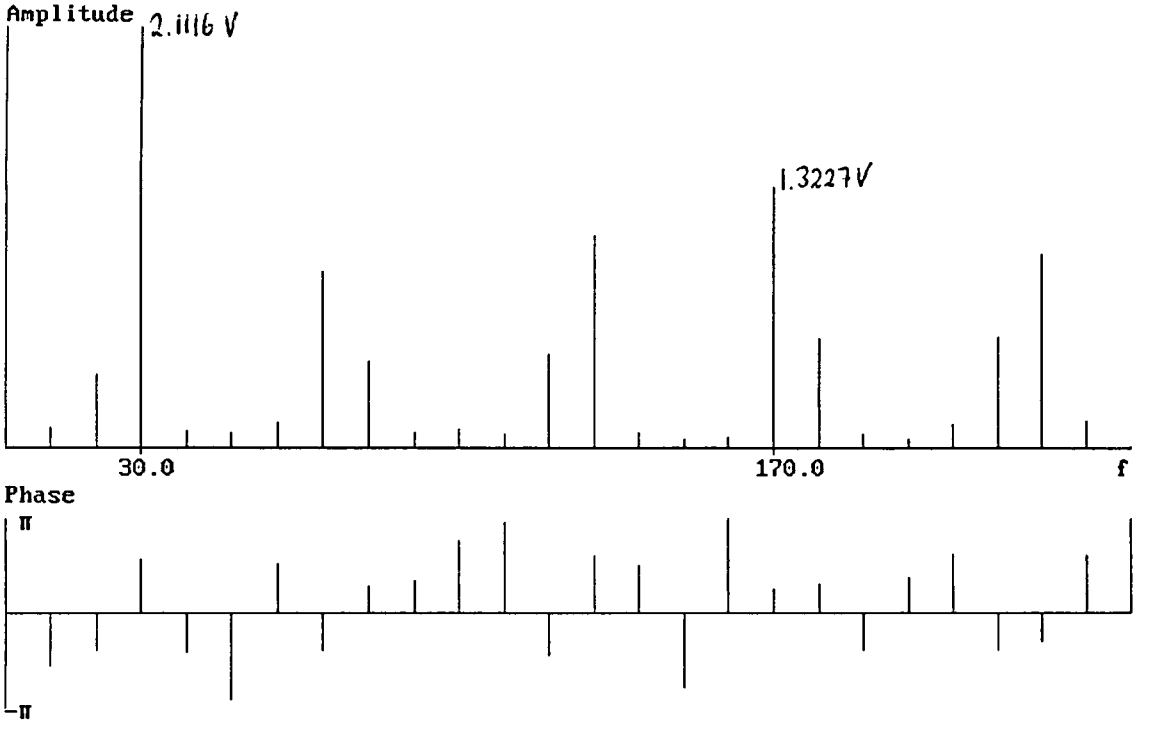


Figure 4.10. Estimated spectrum of example signal sampled by scheme 2.

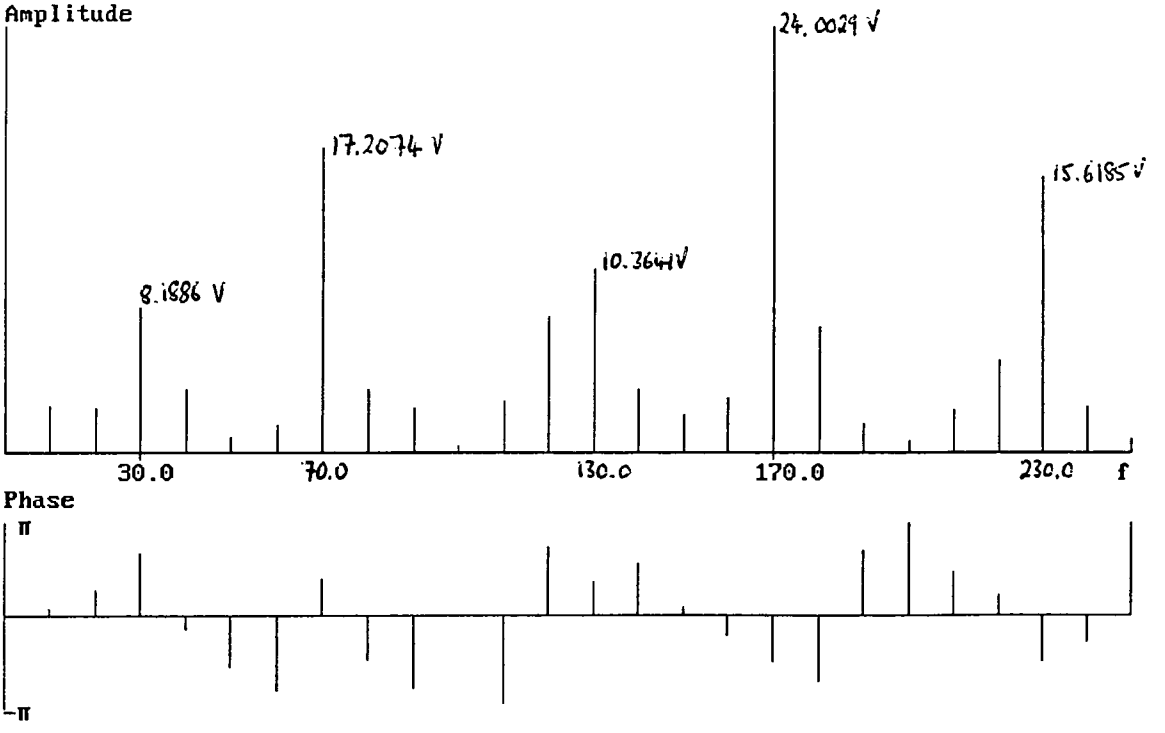


Figure 4.11. Incorrectly 'improved' spectrum of signal after periodic sampling with dither.

The simulations reveal that, in the cases when a signal is sampled using periodic sampling with dither, it is not possible to determine the coefficients a_m and b_m, used to

describe the signal by equation (4.1) for each harmonic at a frequency f_m, without ambiguity from the N sample points. It is possible to do so by using the matrix operation described above, if the signal is sampled using an additive pseudo-random sampling scheme. It is not yet known why this phenomenon occurs for periodic sampling with dither.

The four spectra above are coarse (coefficients are evaluated at frequency steps of $\delta f = 10$ Hz) because the simulation program of appendix C finds it understandably impossible to find the inverse of very large matrices. Evaluating the inverse of a very large matrix takes a long time, requires an excessive amount of storage and involves the use of very large floating-point numbers (sometimes too large for a computer's numerical representation.) For the example 4.6, with a system bandwidth of $B_{pseudo} = 250$ Hz, the inverse of a 50 by 50 matrix was calculated (dimensions given by equation (4.50),) enabling the coefficients to be calculated at only 26 points; ie. at frequencies $f_m = m.\delta f$ for $m = 0$ to 25. This should not be seen as a limitation to the technique but as an inconvenient restriction enforced by hardware limitations.

4.7 Conceptual Interpretation and Discussion of the Technique.

When a band-limited, periodic signal (assumed to be modelled by equation (4.1)) is sampled irregularly by either of the sampling schemes rigidly defined in section 4.3, the time required to acquire N samples of the signal will inevitably be greater than the duration to obtain an equal number of samples by taking the samples at uniform intervals. That is to say, irregular sampling takes a longer time to obtain an equal amount of information (in terms of samples taken) as uniform sampling. This is a direct consequence of irregular sampling operating at a sub-Nyquist rate (for some given bandwidth) relative to uniform sampling. For example, consider a signal band-limited to 250 Hz, sampled at 100 points. This could be sampled by using uniform sampling at 500 Hz or, say, additive pseudo-random sampling with $\tau_1 = 1/100$ s and $\tau_2 = 1/125$ s (corresponding to an average sampling rate of 112.5 Hz.) Using the irregular sampling

scheme, it would take approximately 100/112.5 s to take the required number of samples, whereas it would only take 1/5 s using the uniform sampling scheme.

Furthermore, as with the conventional discrete Fourier transform, some care is needed when using the pseudo-random DFT because it is valid only for the special case of a band-limited periodic signal. The transform will only produce an approximation to the continuous Fourier transform spectrum if there is not an integer number of complete cycles of the input signal sampled over the duration of the time domain window. If the number of cycles sampled in the window interval is incomplete, then discontinuities will exist at the extremities of the interval and the periodic signal will no longer be band-limited. A form of distortion known as leakage will be introduced into the spectrum. The resultant approximated spectrum can be made more accurate only by increasing the window interval for non-periodic signals (by effectively taking more samples); by making the window interval equal to a multiple of the actual period for periodic signals; or by altering the sampling scheme for a much greater bandwidth.

In other words, all the problems that exist with the conventional DFT due to taking a finite length sequence of samples are also applicable to the pseudo-random DFT.

It has been shown that the estimated spectrum produced by the pseudo-random transform may be improved if the signal samples where taken using an additive pseudo-random sampling scheme and hence the original signal may be reconstructed error-free from samples taken at irregular intervals. This is only possible if the sampling instances are known prior to sampling. In most cases, it is expected that the instances will not be known until sampling takes place. Evaluating the inverse of the matrix A every time samples are taken would incur excessive computational overheads and would therefore be impractical for a real-time system. However, as with the conventional DFT, the approximated spectrum can be made more accurate by taking a greater number of samples.

Consider the sampling of a signal containing two harmonics in phase; one at 20 Hz with an amplitude of 1 V, and the other at 90 Hz of amplitude 2 V. Figures 4.12, 4.13 and 4.14 show the signal's estimated amplitude spectrum when samples using a variety of schemes.

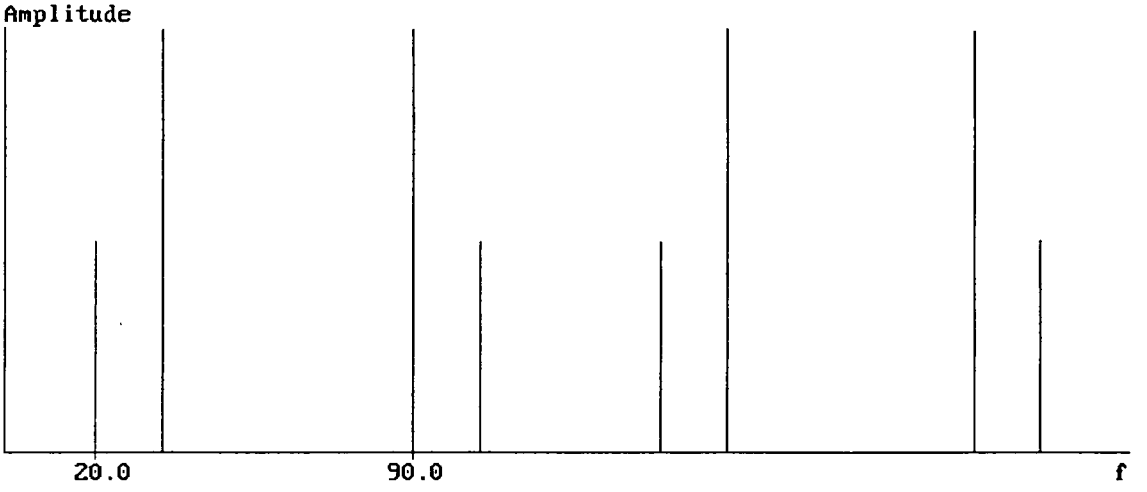


Figure 4.12. 500 samples taken uniformly every 1/125 s.

Figure 4.12 clearly show the aliases that result when uniform sampling is employed.

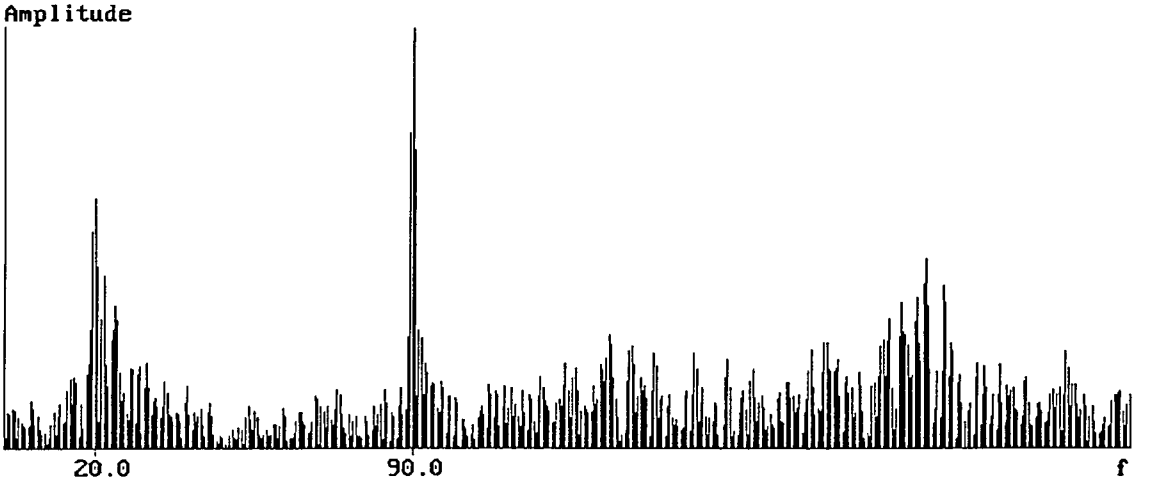


Figure 4.13. 100 samples taken using an additive pseudo-random sampling scheme with τ_1 = 1/100 s and τ_2 = 1/125 s.

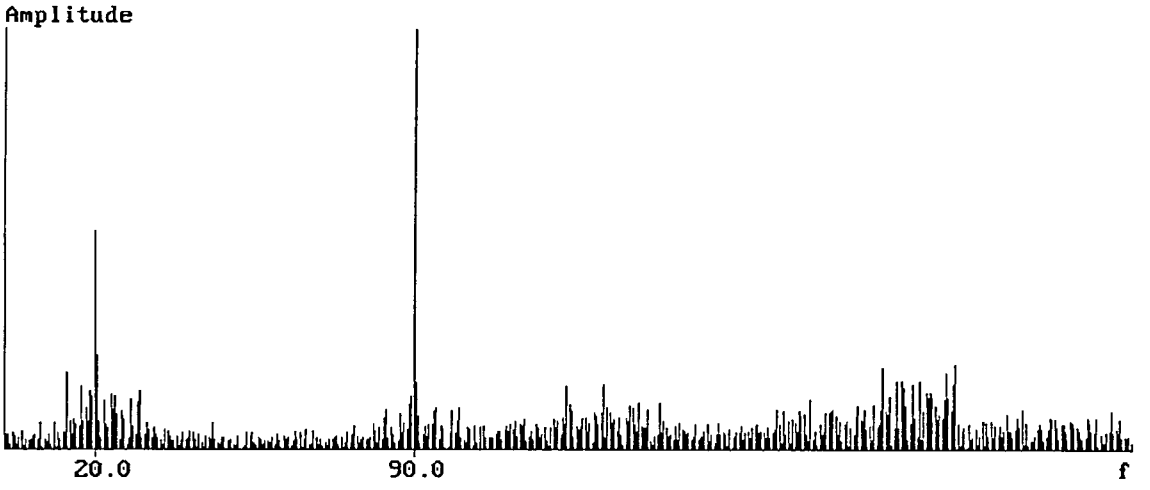


Figure 4.14. 500 samples taken using an additive pseudo-random sampling scheme with $\tau_1 = 1/100$ s and $\tau_2 = 1/125$ s.

Figures 4.13 and 4.14 illustrate how the estimated spectrum is improved by taking a larger number of samples. Furthermore, when compared with figure 4.12, it can be seen that aliases do not appear in the spectra in the form of frequency shifted replicas of the original signal, but spread in the form of broadband noise. The method described in section 4.6 shows that it is possible to unambiguously identify the original signal from this noise when samples are taken using the additive pseudo-random sampling scheme.

A direct realisation of the pseudo-random DFT is clearly excessive in computation for a large number of points, which is required to reduce the noise present in the spectrum of the irregular samples. It is therefore necessary to use a fast algorithm for its implementation. Unfortunately, a decimation in time or a decimation in frequency form **[20, Proakis & Manolakis]** of fast Fourier transform is not applicable, as such an approach requires the input sequence to be evenly divided and makes use of the orthogonality relationships between one 'twiddle factor' and another. It could be advantageous to formulate a fast algorithm to implement the pseudo-random DFT of equation (4.1). However, the technique has some considerable limitations (as does the conventional discrete Fourier transform) but it has been demonstrated that sub-Nyquist sampling is possible by using additive pseudo-random sampling, with a gain in system bandwidth at the expense of signal-to-noise power ratio.

5. DESCRIPTION OF SIMULATION PROGRAMS

The three techniques described in chapters 2, 3 and 4 have been simulated on a PC based system with a floating-point co-processor to show that they have been sufficiently and rigidly defined and to help in verifying the analysis of their characteristics and performance. NAG library routines have been used to determine the characteristics of errors generated by taking the DFT of a sequence of uniformly spaced samples, and then an inverse DFT to reproduce the sequence.

The simulation programs of the three techniques are written in the 'C' programming language and have been compiled using the Microsoft® 'C' optimizing compiler with the "compact" memory model. The simulation program that uses the NAG library is written in the Pascal programming language and was executed on a MTS Mainframe computer system. Their full source codes are listed in the appendices.

The question of "how" the simulation programs work is intentionally not addressed here. It is expected that the reader has sufficient knowledge of 'C' and Pascal to understand the programming with assistance from the comments within the listings and their structure, or should acquire such knowledge. However, "what" the simulation programs do is described.

5.1 Simulation of Single Active Element Dealiasing Algorithm with DFT Errors Considered.

The technique described in chapter 2 is simulated.

The program takes the three sampling frequencies of equation (2.12) as its input and prompts the user as to whether or not the dealiasing algorithm is to be simulated with consideration of the errors imposed by the limited resolution of the discrete Fourier transforms used. If so, the number of points, N used in calculating the DFTs is

requested. The program ensures that the number of points specified satisfies the conditions of equation (2.27).

The operational bandwidth of the algorithm is returned as given by equation (2.16) if DFT limitations are not to be considered, or otherwise by equation (2.24). The program then tests the dealiasing algorithm for every frequency ranging from zero to the pre-calculated maximum operational frequency in steps of some user-entered increment, in an attempt to find an input frequency for which the technique does not work. A plot of the output frequency patterns from the three samplers/filters against input frequency is displayed, and if the DFT limitations are considered, a plot of the error in the frequency as evaluated by the algorithm compared with the actual input frequency is also presented. If the dealiasing algorithm operates correctly, this error plot will show that the error does not exceed the maximum permissible error, as given by equation (2.22), at any time.

The source code for this program is listed in appendix A.

5.2 Simulation of Multiple Active Element Dealiasing Algorithms.

The technique described in chapter 3 is simulated.

The program takes the three sampling frequencies of equation (2.12) as its input and returns the operational bandwidth for the technique as given by equation (2.16). The simulation program prompts the user for a list of frequencies to correspond with the harmonics of the input signal to be modelled. The frequencies entered must be within the operational bandwidth calculated. The program then selects the most appropriate multiple active element dealiasing algorithm (either that of section 3.1 or section 3.2) for the modelled system, by using a criterion based on the relative efficiency of the two algorithms as described in section 3.4.

As output, the program says which algorithm was selected and lists the frequencies of all the harmonics that are believed to be in the input spectrum according to the alias reducing algorithm used. A plot of these frequencies and those of the actual input is produced for comparison, with the remaining 'ghost' frequencies (if any) highlighted.

The source code for this program is listed in appendix B.

5.3 Direct Realisation of the Pseudo-random Discrete Fourier Transform.

This is the most complex of the simulation programs and models every aspect of the technique described in chapter 4.

The program initially requests parameters to describe the sampling scheme to be modelled. This requires the number of sampling points to be taken, the type of sampling (uniform, additive pseudo-random, or periodic with dither,) and relevant sampling frequency data. The input signal bandwidth limitation is calculate and output for the scheme described on the basis of the theory laid out in section 4.4.

Parameters may be entered to describe a complex input signal. The signal to be simulated is assumed to be formed of a summation of any finite number of sinusoidal harmonics of specified frequency, amplitude and relative phase. Sampling of this signal by the scheme declared is simulated, using a pseudo-random number generator with an approximately uniform distribution for the irregularly spaced sampling schemes.

A number of operations may be preformed on the data produced by this process. The pseudo-random discrete Fourier transform of equation (4.9) can be evaluated and displayed at frequencies ranging from zero to a frequency corresponding to the system bandwidth at intervals, δf specified by the user. The pseudo-random transform and its inverse described by equation (4.44) can be determined (with $\delta f = 1/(t_{N-1} + \tau_{min})$,) and

the errors associated with the signal amplitude as determined by the inversion formulae at each sample instance in comparison with the actual signal amplitude are plotted, in addition to the estimated signal spectrum. Alternatively, the matrix method proposed in section 4.6 to improve the estimated Fourier coefficients may be modelled.

The source code for this program is listed in appendix C.

5.4 DFT and Inverse DFT using NAG Library Routines.

In section 4.5, error plots were presented to show the magnitude of the errors generated when taking the pseudo-random DFT of a sequence of samples and then the inverse transform to reproduce the sequence. These plots where generated by executing the simulation program described in section 5.3 on a PC based system with a floating-point co-processor. In order to verify that the magnitude of these errors is consistent with floating-point arithmetic errors, a simulation was also produced using the NAG library routines.

The NAG library routines are available on a MTS Mainframe computer system, which uses a different (and more accurate) representation of floating-point numbers to the PC based system. For this reason, the program listed in appendix D also simulates the pseudo-random DFT and its inverse. The program models uniform sampling of a signal to produce a sequence of consecutive sample values. The NAG library routines are then used to perform a conventional DFT and then an inverse DFT on this sequence to produce a new sequence with error. Similarly, the original sequence of sample values act as input to the pseudo-random DFT and then to its inverse, producing yet another sequence with error. The errors in the two new sequences (one produced by NAG routines and the other by the pseudo-random transforms) are then listed for a direct comparison.

The source code of this program is listed in appendix D.

6. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH.

A rigourous algorithm has been defined to establish the frequency and amplitude of a pure sinusoidal signal (ie. a signal consisting of only its fundamental harmonic) when sampled uniformly by three samplers simultaneously, although the frequency at which these samplers operate may be less than the twice the frequency of the sinusoidal signal. That is, the sinusoidal signal may be sampled at a sub-Nyquist rate and any resultant aliasing may be resolved. The output of each sampler is low-pass filtered 'ideally' with a cut-off at half the sampling frequency of the corresponding sampler. The algorithm has been shown to work within an acceptable and defined tolerance when only an approximation of the frequency of the signals that evolve from the sampling/filtering process, can be made because of limitations with the conventional discrete Fourier transform. The algorithm functions correctly up to a well defined maximum limit of the frequency of the sinusoidal signal, which depends on the three sampling frequencies used.

An attempt to extend the algorithm developed by two analogous, although conceptually diverse methods, so as to eliminate aliases due to sampling (in the same way) a signal containing a sum of harmonics, proved only to reduce the frequency ambiguities. There is the possibility that erroneous harmonics may be identified as part of the signal, even for a signal containing only a few harmonics. It was shown that the only way to optimise the reduction in ambiguities, is to use a very large number of uniform samplers, each succeeded by a low-pass filter. However, the probability of finding erroneous harmonics remains and the excessive computational overheads imposed by the two methods rendered such an approach impractical. In fact, it would be far quicker and more accurate to use sampling at greater than the Nyquist rate. Alternatively, random sampling was proposed.

Irregular sampling of a continuous signal was investigated in terms of the spectrum of the samples taken. The conventional discrete Fourier transform is not applicable to non-uniform samples and so a pseudo-random DFT was developed to determine an estimated spectrum for samples taken at irregular time intervals. Two sampling schemes to generate pseudo-random sampling instances are defined; namely, additive pseudo-random sampling and periodic sampling with dither. The periodicity and symmetry of the transform were found and it is shown that a signal must be band-limited to a well-defined frequency to avoid aliasing for the two sampling schemes. The maximum sampling rate may be well below twice the bandwidth of the limited signal. The transform inverse is derived and shown to function correctly within the floating-point limits of the simulation. If the signal is sampled irregularly using the additive pseudo-random sampling scheme, then its resultant estimated spectrum is consistently alias-free and, if the instances at which each sample was taken were known prior to sampling, the exact Fourier coefficients of the original signal may be determined from it by the matrix method described. The signal must be band-limited to half the pseudo-random transform periodicity, which is dependent upon the irregularly spaced sampling instances.

Simulations of the techniques were made and their results used throughout to illustrate and verify some of the arguments presented.

When samples of a signal are taken at irregular time intervals using an additive pseudo-random sampling scheme, the alias-free spectrum estimated by the pseudo-random DFT shows the presence of noise. It is suggested that further research be undertaken to determine the level of this noise, the parameters upon which it depends (the number of samples taken being one,) and develop methods to reduce or eliminate it. Furthermore, the effects of sample quantization are unknown and should be investigated. The use of pseudo-randomly varying quantization thresholds may be examined to see if quantization noise can be reduced. The estimated spectrum will somehow be affected by random variations in the sampling instances $\{t_k\}$. The effect on the spectrum

estimation due to this or due to using different distributions of pseudo-random numbers in the production of the sampling instances, also requires investigation. Should such further research find the pseudo-random discrete Fourier transform to be of exceptional value, a fast algorithm should be developed to improve the efficiency of its implementation.

Papers with a strong mathematical basis which may be of use in extending the research and which have been helpful from time-to-time to the research already undertaken are listed in the bibliography with a number of items which have been of use throughout the research.

APPENDIX A

Program Source Code for the Simulation of the Single Active Element Dealiasing Algorithm with DFT Errors Considered.

```
/*******************************************
 *                                         *
 * Sub-Nyquist Sampling Algorithm Simulation *
 *                                         *
 * - Single Active Element Analysis -      *
 * - Uniform sampling scheme -             *
 * - DFT errors may be considered -        *
 * - Errors and output patterns plotted -  *
 *                                         *
 *******************************************/

/************************
 *                      *
 * include system files *
 *                      *
 ************************/

#include <stdio.h>
#include <graph.h>
#include <conio.h>
#include <float.h>
#include <math.h>

/*****************************
 *                           *
 * data processing routines *
 *                           *
 *****************************/

/*
 * Return operational bandwidth of system with or without DFT errors considered
 *
 * fs - array of three sampling frequencies
 * n - number of points used to calculate the DFTs. Assumed equal for all three
 *     n = 0 means DFT errors not considered
 */
double frequency_range (fs, n)
double fs[];
int n;
{

    double Bopt, Berr;

    if (n == 0) {
        /* equation (2.16) */
        Bopt = fs[0]*fs[2]/(2.0*(fs[2] - fs[0])) + fs[1]*0.5;
        return (Bopt);
        }
    else {
        /* equation (2.24) */
        Berr = (fs[0]*fs[2]/(fs[2] - fs[0])) * (0.5 - 1.0/(double) n);
        return (Berr);
        }

}
```

```
/*
 * Return maximum permissible error in calculating input frequency
 *
 * fs - array of three sampling frequencies
 * n - number of points used to calculate the DFTs. Assumed equal for all three
 *     n = 0 means DFT errors not considered
 */
double max_error (fs, n)
double fs[];
int n;
{

    double dfxmax;

    if (n == 0)
        return (0.0);
    else {
        /* equation (2.22) */
        dfxmax = fs[0]*fs[2]/((double) n*(fs[2] - fs[0]));
        return (dfxmax);
        }

}

/*
 * Calculate frequency of output signal from the three samples as given by DFT
 *
 * fs - array of three sampling frequencies
 * n - number of points used to calculate the DFTs
 *     n = 0 means DFT errors not considered
 * in - frequency of input signal
 * out - array of frequencies of three output signals
 */
void simulate_output (fs, n, in, out)
double fs[];
int n;
double in;
double out[];
{

    register int i;
    double k, rem, intpart;

    for (i = 0; i <= 2; i++) {
        /* equation (2.13) */
        k = floor (in / fs[i]);
        if ((in >= k*fs[i]) && (in <= (k + 0.5)*fs[i]))
            out[i] = in - k*fs[i];
        else
            out[i] = (k + 1.0)*fs[i] - in;
        if (n != 0) {
            /* truncate output to simulate error given by equation (2.20) */
            rem = modf (out[i]*(double) n/fs[i], &intpart);
            if (rem > 0.5)
                intpart += 1.0;
            out[i] = intpart*fs[i]/(double) n;
            }
        }

}
```

```
/*
 * Dealiasing algorithm
 * Determines the approximate frequency of the signal input from the DFT outputs
 * Returns zero if fails to find a component frequency
 *
 * fs - array of three sampling frequencies
 * n - number of points used to calculate the DFTs
 *     n = 0 means DFT errors not considered
 * out - array of frequencies of three output signals determined by DFTs
 * p_cf - pointer to the correct approximate frequency of the input
 */
int dealias (fs, n, out, p_cf)
double fs[];
int n;
double out[];
double *p_cf;
{

    double idfs1, idfs2, idfs3;
    double r[8][3], dfx, poss_freq;
    register int i, num_found;
    double dr0max, dr1max, dr2max;

    idfs1 = 1.0/(fs[1] - fs[0]);
    idfs2 = 1.0/(fs[2] - fs[1]);
    idfs3 = 1.0/(fs[2] - fs[0]);

    /* calculate cycle count for the eight regions of the output pattern */
    /* equation (2.18) */
    r[7][0] = (out[0] - out[1])*idfs1;
    r[7][1] = (out[1] - out[2])*idfs2;
    r[7][2] = (out[0] - out[2])*idfs3;
    r[1][0] = (fs[0] - out[0] - out[1])*idfs1;
    r[1][1] = r[7][1];
    r[1][2] = (fs[0] - out[0] - out[2])*idfs3;
    r[2][0] = -r[7][0] - 1.0;
    r[2][1] = (fs[1] - out[1] - out[2])*idfs2;
    r[2][2] = r[1][2];
    r[3][0] = r[2][0];
    r[3][1] = -r[7][1] - 1.0;
    r[3][2] = -r[7][2] - 1.0;
    r[4][0] = (out[0] + out[1])*idfs1;
    r[4][1] = -r[7][1];
    r[4][2] = (out[0] + out[2])*idfs3;
    r[5][0] = r[7][0];
    r[5][1] = (out[1] + out[2])*idfs2;
    r[5][2] = r[4][2];
    r[6][0] = r[4][0];
    r[6][1] = r[2][1] + 1.0;
    r[6][2] = (fs[2] - out[2] + out[0])*idfs3;
    r[0][0] = r[1][0];
    r[0][1] = r[5][1];
    r[0][2] = (fs[0] - out[0] + out[2])*idfs3;

    /* use simple algorithm if DFT errors are not to be considered */
    if (n == 0) {
        *p_cf = -1.0;
        num_found = 0;
        for (i = 0; i <= 7; i++)
            if (r[i][0] == r[i][1] && r[i][1] == r[i][2] &&
```

```
            (floor (r[i][2]) == r[i][2])) {
                if (i >= 4)
                    poss_freq = out[0] + r[i][2]*fs[0];
                else
                    poss_freq = (r[i][2] + 1.0)*fs[0] - out[0];
                if (poss_freq != *p_cf &&
                    poss_freq >= 0.0 &&
                    poss_freq < frequency_range (fs, n)) {
                        num_found += 1;
                        *p_cf = poss_freq;
                        }
                }
        }

    /* otherwise use modified algorithm */
    else {

        /* evaluate maximum error in cycle counts - equation (2.21) */
        dr0max = (fs[0] + fs[1])/(2.0 * (fs[1] - fs[0]) * (double) n);
        dr1max = (fs[1] + fs[2])/(2.0 * (fs[2] - fs[1]) * (double) n);
        dr2max = (fs[0] + fs[2])/(2.0 * (fs[2] - fs[0]) * (double) n);

        /* evaluate max error in approximating input frequency */
        dfx = max_error (fs, n);

        /* find valid cycle count and hence the frequency of input signal */
        *p_cf = -2.0*dfx;
        num_found = 0;
        for (i = 0; i <= 7; i++)
            if (fabs (r[i][0] - r[i][1]) < dr0max + dr1max &&
                fabs (r[i][1] - r[i][2]) < dr1max + dr2max &&
                fabs (r[i][0] - r[i][2]) < dr0max + dr2max &&
                (fabs (floor (r[i][0]) - r[i][0]) < dr0max ||
                 fabs (ceil (r[i][0]) - r[i][0]) < dr0max) &&
                (fabs (floor (r[i][1]) - r[i][1]) < dr1max ||
                 fabs (ceil (r[i][1]) - r[i][1]) < dr1max) &&
                (fabs (floor (r[i][2]) - r[i][2]) < dr2max ||
                 fabs (ceil (r[i][2]) - r[i][2]) < dr2max)) {
                    if (i >= 4)
                        poss_freq = out[0] + r[i][2]*fs[0];
                    else
                        poss_freq = (r[i][2] + 1.0)*fs[0] - out[0];
                    if (poss_freq >= 0.0 &&
                        poss_freq < frequency_range (fs, n) + dfx) {
                            num_found = 1;
                            *p_cf = poss_freq;
                            }
                    }
        }

    return (num_found == 1);

}
```

```
/**************************
 *                        *
 * user interface routines *
 *                        *
 **************************/

/*
 * Program introduction display
 */
void heading ()
{

    printf (" [0mThe Analysis of Signals Sampled at a Sub-Nyquist Rate\n\n");
    printf ("P C Bagshaw    December 1989\n\n");
    printf ("Simulation of Dealiasing Algorithm\n");
    printf ("- Single Active Element Analysis -\n");
    printf ("- Uniform sampling scheme -\n");
    printf ("- DFT errors may be considered -\n");
    printf ("- Errors and output patterns plotted -\n\n");

}

/*
 * Request operating frequencies of the three samplers
 *
 * fs - array of three sampling frequencies
 */
void select_sample_frequencies (fs)
double fs[];
{

    register int boolean, i;
    float in_sample_f;

    printf ("Enter three sampling frequencies\n");
    boolean = 1;
    while (boolean) {
        for (i = 0; i <= 2; i++) {
            printf ("fs%d: ", i + 1);
            scanf ("%f", &in_sample_f);
            fs[i] = (double) in_sample_f;
            }
        /* ensure equation (2.12) holds */
        if (boolean = (0.0 >= fs[0] || fs[0] >= fs[1]
                || fs[1] >= fs[2]))
            printf ("error: 0 < fs1 < fs2 < fs3 not satisfied\n");
        }
    return;

}
```

```
/*
 * Request number of points to be used by DFTs and check sufficient are used
 *
 * fs - array of three sampling frequencies
 * p_n - pointer to the number of points used to calculate the DFTs
 */
void read_num_dft_points (fs, p_n)
double fs[];
int *p_n;
{

    register int sufficient_points = 0;
    char key;

    printf ("Are DFT errors to be considered (y/n)? ");
    while ((key = getch ()) != 'y' && key != 'n')
        ;
    printf ("%c\n", key);
    if (key == 'n') {
        *p_n = 0;
        return;
        }
    while (!sufficient_points) {
        printf ("Enter number of points in DFT: ");
        scanf ("%d", p_n);
        /* ensure equation (2.27) holds */
        sufficient_points = ((double) *p_n > (fs[1] + fs[2])/(fs[2] - fs[1]) &&
                             (double) *p_n > (fs[0] + fs[1])/(fs[1] - fs[0]));
        if (!sufficient_points)
            printf ("error: insufficient points for algorithm to operate correctly\n");
        }
    return;

}

/*
 * Plot sampler output frequencies and algorithm error against input frequency
 *
 * s - array of three sampling frequencies
 * n - number of points used to calculate the DFTs
 *     n = 0 means DFT errors not considered
 */
void plot_output_patterns (s, n)
double s[];
int n;
{

    struct videoconfig vc;
    double fr, x_scale, y_scale, fx, fo[3], cf, dfxmax;
    short x;

    if (!set_mode())
        exit (-1);
    _getvideoconfig (&vc);
    _setcolor (_WHITE);
    _settextposition (6, 1);
    printf ("Sampler/Filter outputs: fo1, fo2 and fo3");
    _setlogorg (0, (int) vc.numypixels*0.49);
    fr = frequency_range (s, n);
    dfxmax = max_error (s, n);
```

```
x_scale = (double) (vc.numxpixels - 1)/fr;
_moveto (0, (short) (-s[2]*x_scale/2.0));
_lineto (0, 0);
_lineto (vc.numxpixels, 0);
_moveto (0, 0);
for (fx = 0.0; fx <= fr; fx += 1.0/x_scale) {
    simulate_output (s, n, fx, fo);  /* calculate and plot fo1, fo2, fo3 */
    x = (short) (fx*x_scale);
    _setcolor (12);  /* light red */
    _setpixel (x, (short) (-fo[2]*x_scale));
    _setcolor (10);  /* light green */
    _setpixel (x, (short) (-fo[1]*x_scale));
    _setcolor (9);  /* light blue */
    _setpixel (x, (short) (-fo[0]*x_scale));
    }
_settextposition (1, 1);
printf ("fs1 = %.3f fs2 = %.3f fs3 = %.3f\n", s[0], s[1], s[2]);
printf ("Operational frequency range, B = %f", fr);
_settextposition (16, 1);
printf ("0");
_settextposition (16, 32);
printf ("Input frequency");
_settextposition (16, 80);
printf ("B");

if (n != 0) {
    _settextposition (3, 1);
    printf ("Number of points in DFTs = %d\n", n);
    printf ("Maximum permissible error, ĂdfxmaxĂ = %f\n", dfxmax);
    _settextposition (17, 1);
    printf ("Dealiasing error: ĂdfxĂ");
    _settextposition (18, 1);
    printf ("- %.3f", dfxmax);
    y_scale = -(double) (vc.numypixels - 1)/dfxmax*0.35;
    _setlogorg (0, (int) vc.numypixels*0.93);
    _moveto (0, 0);
    for (fx = 0.0; fx <= fr; fx += 1.0/x_scale) {
        _setcolor (14);  /* yellow */
        simulate_output (s, n, fx, fo);
        if (!dealias (s, n, fo, &cf) || fabs (fx - cf) > dfxmax) {
            /* dealiasing algorithm given incorrect answer */
            _setcolor (13);  /* purple */
            cf = fx - dfxmax;
            }
        _moveto ((short) (fx*x_scale), 0);
        _lineto ((short) (fx*x_scale), (short) (fabs (fx - cf)*y_scale));
        }
    _settextposition (29, 1);
    printf ("0");
    _settextposition (29, 32);
    printf ("Input frequency");
    _settextposition (29, 80);
    printf ("B");
    _setcolor (_WHITE);
    _moveto (0, (short) (dfxmax*y_scale));
    _lineto (0, 0);
    _lineto (vc.numxpixels, 0);
    }

getch ();
```

```
    _setvideomode (_DEFAULTMODE);
    return;

}

/*
 * Select video mode
 */
int set_mode ()
{

    if (_setvideomode (_VRES16COLOR))
        return (_VRES16COLOR);
    if (_setvideomode (_ERESCOLOR))
        return (_ERESCOLOR);
    if (_setvideomode (_MRES16COLOR))
        return (_MRES16COLOR);
    else
        return (0);

}

/****************
 *              *
 * main program *
 *              *
 ****************/

double sample_freqs[3] = {0.0, 0.0, 0.0};

main ()
{

    int num_points;
    double B, dfx_max, freq_inc = 0.0, fx, fo[3], cf;
    float entered_inc;
    register int errors_found = 0;

    heading ();
    select_sample_frequencies (sample_freqs);
    read_num_dft_points (sample_freqs, &num_points);
    B = frequency_range (sample_freqs, num_points);
    printf ("Frequency range: %f\n", B);
    dfx_max = max_error (sample_freqs, num_points);
    printf ("Maximum permissible error in calculating input frequency: %f\n", dfx_max);
    while (freq_inc <= 0.0) {
        printf ("Test dealiasing algorithm over range with frequency increment: ");
        scanf ("%f", &entered_inc);
        freq_inc = (double) entered_inc;
        }
    printf ("Looking for errors in dealiasing algorithm...\n\n");
    for (fx = 0.0; fx < B; fx += freq_inc) {
        simulate_output (sample_freqs, num_points, fx, fo);
        if (!dealias (sample_freqs, num_points, fo, &cf) || fabs (fx - cf) > dfx_max) {
            errors_found += 1;
            if (errors_found == 1)
                printf ("Algorithm in error for input frequency,\n");
            printf("%f gives incorrect answer %f - error = %f\n", fx, cf, fabs(fx -
cf));
            }
```

```
        }
    if (!errors_found)
        printf ("No");
    else
        printf ("%d", errors_found);
    printf (" errors found\n\nPress any key to continue");
    getch ();
    plot_output_patterns (sample_freqs, num_points);
    exit (0);

}
```

APPENDIX B

Program Source Code for the Simulation of the Multiple Active Element Dealiasing Algorithms.

```
/************************************************************
 *                                                          *
 * Sub-Nyquist Sampling Algorithms Simulation               *
 *                                                          *
 * - Multiple Active Element Analysis -                     *
 * - Uniform sampling scheme for three samplers -           *
 * - Chooses most efficient algorithm for case described - *
 *                                                          *
 ************************************************************/

/************************
 *                      *
 * include system files *
 *                      *
 ************************/

#include <malloc.h>
#include <stdio.h>
#include <graph.h>
#include <conio.h>
#include <float.h>
#include <math.h>

/************************************************************
 *                                                          *
 * define abstract data type for ordered list of frequencies *
 *                                                          *
 ************************************************************/

typedef struct freq_list {
    double frequency;
    struct freq_list *next_frequency;
    } FREQ_LIST;

void add_to_list (p_list_hd, f_value)
FREQ_LIST **p_list_hd;
double f_value;
{
    FREQ_LIST *new_node, *node = NULL, *old_node = NULL;
    register new_value = 1;

    if (*p_list_hd != NULL) {
        node = *p_list_hd;
        while ((new_value = f_value != node->frequency) &&
            node != NULL && f_value > node->frequency) {
                old_node = node;
                node = node->next_frequency;
                }
        }
    if (new_value) {
        new_node = (FREQ_LIST *) malloc (sizeof (FREQ_LIST));
        if (new_node == NULL) {
            fprintf (stderr, "error: insufficient memory available\n");
            exit (-1);
            }
        new_node->frequency = f_value;
        if (*p_list_hd == NULL) {
            *p_list_hd = new_node;
            new_node->next_frequency = NULL;
            }
```

```
        else {
            if (old_node == NULL)
                *p_list_hd = new_node;
            else
                old_node->next_frequency = new_node;
            new_node->next_frequency = node;
            }
        }

}

/*****************************
 *                           *
 * data processing routines *
 *                           *
 *****************************/

double frequency_range (fs)
double fs[];
{

    double Bopt;

    /* equation (2.16) */
    Bopt = fs[0]*fs[2]/(2.0*(fs[2] - fs[0])) + fs[1]*0.5;
    return (Bopt);

}

void simulate_output (fs, in, out)
double fs[];
double in;
double out[];
{

    register int i;
    double k;

    for (i = 0; i <= 2; i++) {
        /* equation (2.13) */
        k = floor (in / fs[i]);
        if ((in >= k*fs[i]) && (in <= (k + 0.5)*fs[i]))
            out[i] = in - k*fs[i];
        else
            out[i] = (k + 1.0)*fs[i] - in;
        }
    return;

}

int single_dealias (fs, out, p_cf)
double fs[];
double out[];
double *p_cf;
{

    double idfs1, idfs2, idfs3;
    double r[8][3], dfx, poss_freq;
    register int i, num_found;
    double dr0max, dr1max, dr2max;
```

```
    idfs1 = 1.0/(fs[1] - fs[0]);
    idfs2 = 1.0/(fs[2] - fs[1]);
    idfs3 = 1.0/(fs[2] - fs[0]);

    /* calculate cycle count for the eight regions of the output pattern */
    /* equation (2.18) */
    r[7][0] = (out[0] - out[1])*idfs1;
    r[7][1] = (out[1] - out[2])*idfs2;
    r[7][2] = (out[0] - out[2])*idfs3;
    r[1][0] = (fs[0] - out[0] - out[1])*idfs1;
    r[1][1] = r[7][1];
    r[1][2] = (fs[0] - out[0] - out[2])*idfs3;
    r[2][0] = -r[7][0] - 1.0;
    r[2][1] = (fs[1] - out[1] - out[2])*idfs2;
    r[2][2] = r[1][2];
    r[3][0] = r[2][0];
    r[3][1] = -r[7][1] - 1.0;
    r[3][2] = -r[7][2] - 1.0;
    r[4][0] = (out[0] + out[1])*idfs1;
    r[4][1] = -r[7][1];
    r[4][2] = (out[0] + out[2])*idfs3;
    r[5][0] = r[7][0];
    r[5][1] = (out[1] + out[2])*idfs2;
    r[5][2] = r[4][2];
    r[6][0] = r[4][0];
    r[6][1] = r[2][1] + 1.0;
    r[6][2] = (fs[2] - out[2] + out[0])*idfs3;
    r[0][0] = r[1][0];
    r[0][1] = r[5][1];
    r[0][2] = (fs[0] - out[0] + out[2])*idfs3;

    /* use simple algorithm since no DFT errors are considered */
    *p_cf = -1.0;
    num_found = 0;
    for (i = 0; i <= 7; i++)
        if (r[i][0] == r[i][1] && r[i][1] == r[i][2] &&
            (floor (r[i][2]) == r[i][2])) {
                if (i >= 4)
                    poss_freq = out[0] + r[i][2]*fs[0];
                else
                    poss_freq = (r[i][2] + 1.0)*fs[0] - out[0];
                if (poss_freq != *p_cf &&
                    poss_freq >= 0.0 &&
                    poss_freq < frequency_range (fs)) {
                        num_found += 1;
                        *p_cf = poss_freq;
                        }
                }

    return (num_found == 1);

}
```

```
/*
 * Implementation of multiple active element ambiguity reduction algorithm no.1
 */
void multiple_dealias_1 (s, o_hd, p_out_hd)
double s[];
FREQ_LIST *o_hd[];
FREQ_LIST **p_out_hd;
{
    double fo[3], cf, fr;
    FREQ_LIST *p_n1, *p_n2, *p_n3;

    p_n1 = o_hd[0];
    while (p_n1 != NULL) {
        fo[0] = p_n1->frequency;
        p_n2 = o_hd[1];
        while (p_n2 != NULL) {
            fo[1] = p_n2->frequency;
            p_n3 = o_hd[2];
            while (p_n3 != NULL) {
                fo[2] = p_n3->frequency;
                if (single_dealias (s, fo, &cf))
                    add_to_list (p_out_hd, cf);
                p_n3 = p_n3->next_frequency;
                }
            p_n2 = p_n2->next_frequency;
            }
        p_n1 = p_n1->next_frequency;
        }
    return;

}

/*
 * Implementation of multiple active element ambiguity reduction algorithm no.2
 */
void multiple_dealias_2 (s, o_hd, p_out_hd)
double s[];
FREQ_LIST *o_hd[];
FREQ_LIST **p_out_hd;
{

    FREQ_LIST   *p_n[3], *set[3];
    register int n, control, other[2];
    double sf;

    for (control = 0; control <=2; control++) {
        set[control] = NULL;
        p_n[control] = o_hd[control];
        while (p_n[control] != NULL) {
            sf = 0.0;
            n = 0;
            while ((sf = (double) (n++)*s[control] + p_n[control]->frequency)
                < frequency_range(s))
                    add_to_list (&set[control], sf);
            sf = 0.0;
            n = 1;
            while ((sf = (double) (n++)*s[control] - p_n[control]->frequency)
                < frequency_range(s))
                    add_to_list (&set[control], sf);
            p_n[control] = p_n[control]->next_frequency;
```

```
            }
        }
    while (set[0] != NULL && set[1] != NULL && set[2] != NULL) {
        if (set[2]->frequency >= set[1]->frequency)
            if (set[2]->frequency >= set[0]->frequency) {
                other[0] = 0;
                other[1] = 1;
                control = 2;
                }
            else {
                control = 0;
                other[0] = 1;
                other[1] = 2;
                }
        else
            if (set[1]->frequency >= set[0]->frequency) {
                other[0] = 0;
                control = 1;
                other[1] = 2;
                }
            else {
                control = 0;
                other[0] = 1;
                other[1] = 2;
                }
        for (n = 0; n <= 1; n++)
            while (set[other[n]] != NULL &&
                set[other[n]]->frequency < set[control]->frequency)
                    set[other[n]] = set[other[n]]->next_frequency;
        if (set[other[0]] != NULL && set[other[1]] != NULL)
            if (set[0]->frequency == set[1]->frequency &&
                set[1]->frequency == set[2]->frequency) {
                    add_to_list (p_out_hd, set[0]->frequency);
                    set[control] = set[control]->next_frequency;
                    }
        }
    return;

}

/***************************
 *                         *
 * user interface routines *
 *                         *
 ***************************/

void print_heading ()
{

    printf (" [0mThe Analysis of Signals Sampled at a Sub-Nyquist Rate\n\n");
    printf ("P C Bagshaw    January 1990\n\n");
    printf ("Simulation of Dealiasing Algorithms\n");
    printf ("- Multiple Active Element Analysis -\n");
    printf ("- Uniform sampling scheme for three samplers -\n");
    printf ("- Chooses most efficient algorithm for case described -\n\n");
    return;

}
```

```
void select_sample_frequencies (fs)
double fs[];
{

    register int boolean, i;
    float in_sample_f;

    printf ("Enter three sampling frequencies\n");
    boolean = 1;
    while (boolean) {
        for (i = 0; i <= 2; i++) {
            printf ("fs%d: ", i + 1);
            scanf ("%f", &in_sample_f);
            fs[i] = (double) in_sample_f;
            }
        /* ensure equation (2.12) holds */
        if (boolean = (0.0 >= fs[0] || fs[0] >= fs[1]
                || fs[1] >= fs[2]))
            printf ("error: 0 < fs1 < fs2 < fs3 not satisfied\n");
        }
    return;

}

void plot_analysis (s, in_hd, out_hd)
double s[];
FREQ_LIST *in_hd, *out_hd;
{

    struct videoconfig vc;
    double fr, scale, text_scale;
    register short y;
    FREQ_LIST *node, *np;
    register int alias;

    if (!set_mode())
        exit (-1);
    _getvideoconfig (&vc);
    fr = frequency_range (s);
    scale = (double) (vc.numxpixels-1)/fr;
    text_scale = 80.0/ceil (fr);
    _settextposition (1, 1);
    _outtext ("Multiple Active Element Signal Analysis ");
    _outtext ("using sub-Nyquist Dealiasing Algorithm\n");
    printf ("fs1 = %.3f fs2 = %.3f fs3 = %.3f\n", s[0], s[1], s[2]);
    printf ("Optimum operational frequency range, B = %f", fr);
    _setcolor (15);
    _settextposition (13, 1);
    printf ("0");
    _settextposition (13, 80);
    printf ("B");
    _settextposition (14, 1);
    _outtext ("Frequency components of input signal");
    _setlogorg (0, (short) vc.numypixels*0.39);
    _moveto (0, y = (short) -vc.numypixels*0.2);
    _lineto (0, 0);
    _lineto (vc.numxpixels, 0);
    _moveto (0, 0);
    node = in_hd;
    while (node != NULL) {
```

```
        _setcolor (10); /* light green */
        _moveto ((short) (node->frequency*scale), 0);
        _lineto ((short) (node->frequency*scale), y);
        _settextposition (13, 1 + (int) (text_scale*ceil (node->frequency)));
        printf ("%.0f", node->frequency);
        node = node->next_frequency;
        }
    _setcolor (15);
    _settextposition (25, 1);
    printf ("0");
    _settextposition (25, 80);
    printf ("B");
    _settextposition (26, 1);
    _outtext ("Frequency components of output signal");
    _setlogorg (0, (short) vc.numypixels*0.79);
    _moveto (0, y = (short) -vc.numypixels*0.2);
    _lineto (0, 0);
    _lineto (vc.numxpixels, 0);
    _moveto (0, 0);
    node = out_hd;
    while (node != NULL) {
        alias = 1;
        np = in_hd;
        while (alias && np != NULL) {
            alias = node->frequency != np->frequency;
            np = np->next_frequency;
            }
        if (alias)
            _setcolor (12); /* light red */
        else
            _setcolor (14); /* yellow */
        _moveto ((short) (node->frequency*scale), 0);
        _lineto ((short) (node->frequency*scale), y);
        _settextposition (25, 1 + (int) (text_scale*ceil (node->frequency)));
        printf ("%.0f", node->frequency);
        node = node->next_frequency;
        }
    getch ();
    _setvideomode (_DEFAULTMODE);
    return;

}

int set_mode ()
{

    if (_setvideomode (_VRES16COLOR))
        return (_VRES16COLOR);
    if (_setvideomode (_ERESCOLOR))
        return (_ERESCOLOR);
    if (_setvideomode (_MRES16COLOR))
        return (_MRES16COLOR);
    else
        return (0);

}
```

```
/****************
 *              *
 * main program *
 *              *
 ****************/

double      fs[3] = {0.0 ,0.0, 0.0};
FREQ_LIST   *fo_hd[3] = {NULL, NULL, NULL};

main ()
{

    int num_harmonics = 0;
    double fo[3], fr;
    float entered_freq;
    FREQ_LIST *input_freq_hd = NULL, *output_freq_hd = NULL;
    FREQ_LIST *p_node;

    print_heading ();
    select_sample_frequencies (fs);
    printf ("Frequency range: %f\n", fr = frequency_range(fs));
    printf ("\nEnter frequencies of the signal elements, ");
    printf ("terminating with an out-of-range value\n* ");
    scanf ("%f", &entered_freq);
    while (entered_freq >= 0.0 && entered_freq < fr) {
        add_to_list (&input_freq_hd, (double) entered_freq);
        num_harmonics++;
        printf ("* ");
        scanf ("%f", &entered_freq);
        }
    p_node = input_freq_hd;
    while (p_node != NULL) {
        simulate_output (fs, p_node->frequency, fo);
        add_to_list (&fo_hd[0], fo[0]);
        add_to_list (&fo_hd[1], fo[1]);
        add_to_list (&fo_hd[2], fo[2]);
        p_node = p_node->next_frequency;
        }
    /* choose the most efficient algorithm */
    if (num_harmonics*num_harmonics < fr/fs[1]) {
        printf("Running dealiasing algorithm #1...\n");
        multiple_dealias_1 (fs, fo_hd, &output_freq_hd);
        }
    else {
        printf("Running dealiasing algorithm #2...\n");
        multiple_dealias_2 (fs, fo_hd, &output_freq_hd);
        }
    printf ("\nFrequencies believed to be in input spectra after analysis:\n");
    p_node = output_freq_hd;
    while (p_node != NULL) {
        printf ("* %f\n", p_node->frequency);
        p_node = p_node->next_frequency;
        }
    printf ("Press any key to continue");
    getch ();
    plot_analysis (fs, input_freq_hd, output_freq_hd);
    exit (0);

}
```

APPENDIX C

Program Source Code for the Direct Realisation of the Pseudo-random Discrete Fourier Transform.

```
/*******************************************
 *                                         *
 * Pseudo-Random Fourier Transform Simulator *
 *                                         *
 *******************************************/

/************************
 *                      *
 * include system files *
 *                      *
 ************************/

#include <string.h>
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <malloc.h>
#include <io.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <graph.h>
#include <limits.h>
#include <errno.h>

/***************************
 *                         *
 * define global constants *
 *                         *
 ***************************/

#define PI                  3.14159265358979323849
#define TWOPI               6.28318530717958647698
/* maximum value 'double' variable can be expressed as an 'unsigned long int' */
#define DBL_LNG_MAX         4294967295.0
#define MAX_FILENAME_SIZE   50

/*******************************
 *                             *
 * error message handling      *
 *                             *
 * void error (int err_type); *
 *                             *
 *******************************/

#define NO_MEM          0
#define UNDEF_TYPE      1
#define NOT_RATIONAL    2
#define OVERFLOW        3
#define FILE_ERR        4
#define SINGULAR_MATRIX 5
#define NO_TEMP         6
#define NO_SF           7
#define NO_INPUT        8
#define GRAPHICS_MODE   9
#define EOF_FOUND       10
```

```
void error (err_type)
const unsigned int err_type;
{

    char prog[7]; /* program file name */

    strcpy (prog, "pseudo");
    fprintf (stderr, " [1;31m%s: ", prog);
    switch (err_type) {
        case NO_MEM:
            fprintf (stderr, "insufficient memory available");
            break;
        case UNDEF_TYPE:
            fprintf (stderr, "undefined list type");
            break;
        case NOT_RATIONAL:
            fprintf (stderr, "cannot express number in rational form");
            break;
        case OVERFLOW:
            fprintf (stderr, "overflow encountered in calculating system bandwidth");
            break;
        case FILE_ERR:
            perror (NULL);
            break;
        case SINGULAR_MATRIX:
            fprintf (stderr, "singular matrix");
            break;
        case NO_TEMP:
            fprintf (stderr, "cannot create a temporary file name");
            break;
        case NO_SF:
            fprintf (stderr, "no sampling frequency defined");
            break;
        case NO_INPUT:
            fprintf (stderr, "no input signal defined");
            break;
        case GRAPHICS_MODE:
            fprintf (stderr, "cannot open graphics screen");
            break;
        case EOF_FOUND:
            fprintf (stderr, "unexpected end of file encountered");
            break;
        default:
            fprintf (stderr, "undefined error, %u occurred", err_type);
            break;
    }
    fprintf (stderr, " [1;37m\n");
    exit (-1);

}
```

```
/*********************************************************
 *                                                       *
 * define abstract data type for a list of various types  *
 *                                                       *
 * void initialise_list (LIST *p_list, int type_of_list); *
 * void add_to_list (LIST *p_list, ITEM data);           *
 * void rewind_list (LIST *p_list);                      *
 * int read_list (LIST *p_list, ITEM *p_data);           *
 *                                                       *
 *********************************************************/

/*
 * list_type values
 */

#define FREQ_LIST              1 /* signal harmonics data */
#define LIST_SAMPLE_FREQS      2 /* details of possible sampling frequencies */
#define SAMPLE_DATA            3 /* chronological list of sample values and times taken*/
#define FOURIER_TRANSFORM_DATA 4 /* transform output information */

/*
 * define abstract data type
 */

typedef union {
    struct {                                /* FREQ_LIST */
        double frequency, amplitude, phase;
        } element;
    double frequency;                       /* LIST_SAMPLE_FREQS */
    struct {                                /* SAMPLE_DATA */
        double value, instance;
        } sample;
    struct {                                /* FOURIER_TRANSFORM_DATA */
        double fn, real, imaginary;
        } bin;
    } ITEM;

typedef struct body_of_list {
    ITEM item_listed;
    struct body_of_list *next_item;
    } BODY_OF_LIST;

typedef struct {
    int list_type;
    BODY_OF_LIST *list_hd, *cursor;
    } LIST;

/*
 * start up values to be taken by any newly defined variable of type LIST
 */

void initialise_list (p_list, type_of_list)
LIST *p_list;
const int type_of_list;
{

    p_list->list_type = type_of_list;
    p_list->list_hd = NULL;
    p_list->cursor = NULL;
```

```
}

/*
 * append list to include new item
 */

void add_to_list (p_list, data)
LIST *p_list;
ITEM data;
{

    BODY_OF_LIST *new_node, *node = NULL, *old_node = NULL;
    register int new_value = 1;

    if (p_list->list_type == FREQ_LIST ||
        p_list->list_type == LIST_SAMPLE_FREQS) /* unrepeated items required */
        for (node = p_list->list_hd; new_value && node != NULL;
             old_node = node, node = node->next_item) {
            if (p_list->list_type == FREQ_LIST)
                new_value &= (data.element.frequency != node->item_listed.element.frequency);
            if (p_list->list_type == LIST_SAMPLE_FREQS)
                new_value &= (data.frequency != node->item_listed.frequency);
            }
    else /* add new item to end of list */
        old_node = p_list->cursor;
    if (new_value) {
        new_node = (BODY_OF_LIST *) malloc (sizeof (BODY_OF_LIST));
        if (new_node == NULL) error (NO_MEM);
        switch (p_list->list_type) {
            case FREQ_LIST:
                new_node->item_listed.element.frequency = data.element.frequency;
                new_node->item_listed.element.amplitude = data.element.amplitude;
                new_node->item_listed.element.phase = data.element.phase;
                break;
            case LIST_SAMPLE_FREQS:
                new_node->item_listed.frequency = data.frequency;
                break;
            case SAMPLE_DATA:
                new_node->item_listed.sample.value = data.sample.value;
                new_node->item_listed.sample.instance = data.sample.instance;
                break;
            case FOURIER_TRANSFORM_DATA:
                new_node->item_listed.bin.fn = data.bin.fn;
                new_node->item_listed.bin.real = data.bin.real;
                new_node->item_listed.bin.imaginary = data.bin.imaginary;
                break;
            default:
                error (UNDEF_TYPE);
                break;
            }
        if (p_list->list_hd == NULL)
            p_list->list_hd = new_node;
        else
            old_node->next_item = new_node;
        new_node->next_item = NULL;
        p_list->cursor = new_node;
        }
    if (!new_value && p_list->list_type == FREQ_LIST) {
        /* modify parameters if harmonic already listed */
        old_node->item_listed.element.amplitude = data.element.amplitude;
```

```
        old_node->item_listed.element.phase = data.element.phase;
        }

}

/*
 * reset list cursor to the beginning of the list
 */

void rewind_list (p_list)
LIST *p_list;
{

    p_list->cursor = p_list->list_hd;

}

/*
 * read item at list cursor and forward cursor by one item
 * returns 0 if no item
 */

int read_list (p_list, p_data)
LIST *p_list;
ITEM *p_data;
{

    if (p_list->cursor == NULL) /* cursor at end of list */
        return (0);
    else {
        switch (p_list->list_type) {
            case FREQ_LIST:
                p_data->element.frequency = p_list->cursor->item_listed.element.frequency;
                p_data->element.amplitude = p_list->cursor->item_listed.element.amplitude;
                p_data->element.phase = p_list->cursor->item_listed.element.phase;
                break;
            case LIST_SAMPLE_FREQS:
                p_data->frequency = p_list->cursor->item_listed.frequency;
                break;
            case SAMPLE_DATA:
                p_data->sample.value = p_list->cursor->item_listed.sample.value;
                p_data->sample.instance = p_list->cursor->item_listed.sample.instance;
                break;
            case FOURIER_TRANSFORM_DATA:
                p_data->bin.fn = p_list->cursor->item_listed.bin.fn;
                p_data->bin.real = p_list->cursor->item_listed.bin.real;
                p_data->bin.imaginary = p_list->cursor->item_listed.bin.imaginary;
                break;
            default:
                error (UNDEF_TYPE);
                break;
            }
        p_list->cursor = p_list->cursor->next_item;
        return (1);
        }

}
```

```
void free_list (p_list)
LIST *p_list;
{

    BODY_OF_LIST *next;

    while (p_list->list_hd != NULL) {
        next = p_list->list_hd->next_item;
        free ((BODY_OF_LIST *) p_list->list_hd);
        p_list->list_hd = next;
        }
    p_list->cursor = NULL;

}

/*****************************************************************************
 *                                                                           *
 * little helpful maths routines                                             *
 *                                                                           *
 * int even (int number);                                                    *
 * void convert_complex (double a, double b, double *p_amp, double *p_phi); *
 * double sine (double x);                                                   *
 * double cosine (double x);                                                 *
 *                                                                           *
 *****************************************************************************/

int even (number)
int number;
{

    return (number % 2 == 0);

}

void convert_complex (a, b, p_amp, p_phi)
double a, b, *p_amp, *p_phi;
{

    *p_amp = sqrt (a*a + b*b);
    if (fabs (a) < 3e-14 && fabs(b) < 3e-14)
        *p_phi = 0.0;
    else
        *p_phi = atan2 (-b, a);

}

/*
 * slightly more accurate sine and cosine functions
 */

double sine (x)
double x;
{

    x = fmod (x, TWOPI);
    return (sin (x));

}
```

```
double cosine (x)
double x;
{

    x = fmod (x, TWOPI);
    return (cos (x));

}

/*****************************************************************************
 *                                                                           *
 * routines to simulate sample generator - uniform, additive pseudo-random   *
 *                                        or periodic with dither            *
 *                                                                           *
 * int equal_schemes (SAMPLING_SCHEME s1, SAMPLING_SCHEME s2);               *
 * void calc_bandwidth (SAMPLING_SCHEME *p_sampler);                         *
 * void random_sampler (SAMPLING_SCHEME *p_sampler, LIST in_freqs, LIST *p_out_data); *
 *                                                                           *
 *****************************************************************************/

/*
 * abstract data type for sampling scheme parameters
 */

#define UNIFORM     'u'
#define ADDITIVE    'a'
#define DITHER      'd'

typedef struct {
    char scheme; /* to take the value UNIFORM, ADDITIVE or DITHER */
    int num_fs;
    LIST fs;
    int num_points;
    double bandwidth;
    double window_width;
    } SAMPLING_SCHEME;

/*
 * test if two schemes are equivalent other than list of sampling frequencies
 */

int equal_schemes (s1, s2)
SAMPLING_SCHEME s1, s2;
{

    int equal;

    equal = (s1.scheme == s2.scheme &&
             s1.num_fs == s2.num_fs &&
             s1.num_points == s2.num_points &&
             s1.bandwidth == s2.bandwidth &&
             s1.window_width == s2.window_width);
    return (equal);

}
```

```
/*
 * evaluate operational bandwidth of system for specified pseudo-random
 * sampling scheme - equation (4.31)
 */

void calc_bandwidth (p_sampler)
SAMPLING_SCHEME *p_sampler;
{

    ITEM s;
    double lcm, gcd, a, b, old_a, old_b, tmp, int_part, req_fs;
    unsigned long int m, n, r;
    int first_time = 1;

    if (p_sampler->num_fs == 1) { /* only one sampling frequency */
        rewind_list (&p_sampler->fs);
        read_list (&p_sampler->fs, &s);
        p_sampler->bandwidth = s.frequency / 2.0;
        return;
        }
    lcm = 1.0;
    rewind_list (&p_sampler->fs);
    while (read_list (&p_sampler->fs, &s)) {
        /* express number in some rational form, a/b */
        if (p_sampler->scheme == DITHER)
            if (first_time) {
                req_fs = s.frequency;
                a = 1.0;
                b = s.frequency;
                }
            else {
                a = s.frequency;
                b = req_fs;
                }
        else {
            a = 1.0;
            b = s.frequency;
            }
        old_a = a;
        old_b = b;
        while ((modf (a, &tmp) > 1e-10 || modf (b, &int_part) > 1e-10) &&
               a <= DBL_LNG_MAX && b <= DBL_LNG_MAX) {
            a += old_a;
            b += old_b;
            }
        if (a > DBL_LNG_MAX || b > DBL_LNG_MAX)
            /* number cannot be expressed in a rational form within the computer limits */
            error (NOT_RATIONAL);
        /* find greatest common divisor of denominator and numerator to express
           rational number in most optimum form */
        m = (unsigned long int) a;
        n = (unsigned long int) b;
        while ((r = n % m) != 0L) {
            n = m;
            m = r;
            }
        a /= m;
        b /= m;
        /* find greatest common divisor of numerators */
        if (first_time)
```

```
            gcd = a;
        else {
            m = (unsigned long int) gcd;
            n = (unsigned long int) a;
            while ((r = n % m) != 0L) {
                n = m;
                m = r;
                }
            gcd = m;
            }
        /* find greatest common divisor of lcm and denominator */
        if (lcm > DBL_LNG_MAX) error (OVERFLOW);
        m = (unsigned long int) lcm;
        n = (unsigned long int) b;
        while ((r = n % m) != 0L) {
            n = m;
            m = r;
            }
        /* calculate least common multiple */
        lcm *= b/m;
        first_time = 0;
        }
    p_sampler->bandwidth = lcm / gcd / 2.0;

}

/*
 * generate a pseudo-random number between 0 and 1
 */

typedef enum {RESET, CONT} RAND_STATUS;

double random (ctrl)
RAND_STATUS ctrl;
{

    static double rnd = 12357.0;

    if (ctrl == RESET) {
        rnd = 12357.0;
        return (0.0);
        }
    else {
        rnd = fmod ((2045.0*rnd + 1.0), 1048576.0);
        return ((rnd + 1.0)/1048577.0);
        }

}

/*
 * simulate pseudo-random sampler
 */

void random_sampler (p_sampler, in_freqs, p_out_data)
SAMPLING_SCHEME *p_sampler;
LIST in_freqs, *p_out_data;
{

    int i, r, r2;
    ITEM itm;
```

```
double argu, xr, tr = 0.0, dtr = 0.0, old_dtr, fs, fs_max;

random (RESET);
for (i = 0; i < p_sampler->num_points; i++) {
    xr = 0.0;
    rewind_list (&in_freqs);
    while (read_list (&in_freqs, &itm)) {
        argu = TWOPI*itm.element.frequency*tr + itm.element.phase;
        xr += itm.element.amplitude * cosine (argu);
        }
    itm.sample.value = xr;
    itm.sample.instance = tr;
    add_to_list (p_out_data, itm);
    rewind_list (&p_sampler->fs);
    switch (p_sampler->scheme) {
        case UNIFORM:
            read_list (&p_sampler->fs, &itm);
            dtr = 1.0/itm.frequency;
            tr += dtr;
            break;
        case ADDITIVE:
            for (r = (int) floor (random (CONT)*p_sampler->num_fs) + 1;
                 r-- != 0; read_list (&p_sampler->fs, &itm))
                ;
            dtr = 1.0/itm.frequency;
            tr += dtr;
            break;
        case DITHER:
            read_list (&p_sampler->fs, &itm);
            fs = itm.frequency;
            old_dtr = dtr;
            r = (int) floor (random (CONT)*p_sampler->num_fs) + 1;
            if (p_sampler->num_fs % 2 && r == p_sampler->num_fs)
                dtr = 0.0;
            else {
                for (r2 = (r + 1)/2; r2-- != 0; read_list (&p_sampler->fs, &itm))
                    ;
                dtr = itm.frequency/fs;
                dtr *= (r % 2) ? -1.0 : +1.0;
                }
            tr += -old_dtr + 1.0/fs + dtr;
            break;
        default:
            break;
        }
    }
/* determine minimum possible chance in sampling instances */
rewind_list (&p_sampler->fs);
switch (p_sampler->scheme) {
    case UNIFORM:
        read_list (&p_sampler->fs, &itm);
        fs_max = itm.frequency;
        break;
    case ADDITIVE:
        read_list (&p_sampler->fs, &itm);
        fs_max = itm.frequency;
        while (read_list (&p_sampler->fs, &itm))
            fs_max = (itm.frequency > fs_max) ? itm.frequency : fs_max;
        break;
    case DITHER:
```

```
            read_list (&p_sampler->fs, &itm);
            fs = itm.frequency;
            fs_max = 0.0;
            while (read_list (&p_sampler->fs, &itm))
                fs_max = (fs/itm.frequency > fs_max) ? fs/itm.frequency : fs_max;
            fs_max = fs*(1.0 - 2.0*fs_max);
            break;
        default:
            break;
        }
    p_sampler->window_width = tr - dtr + 1.0/fs_max;

}

/*****************************************************************************************
 *                                                                                       *
 * data processing routines                                                              *
 *                                                                                       *
 * void dprft (SAMPLING_SCHEME sampler, double deltaf, LIST sdata, LIST *p_ftdata);      *
 * void inv_dprft (SAMPLING_SCHEME sampler, LIST ftdata, LIST sdata, LIST *p_idft_data); *
 *                                                                                       *
 *****************************************************************************************/

/*
 * perform direct discrete pseudo-random Fourier transform - equation (4.9)
 */

void dprft (sampler, deltaf, sdata, p_ftdata)
SAMPLING_SCHEME sampler;
double deltaf;
LIST sdata, *p_ftdata;
{

    unsigned long int num_bins, evaluate, count, i;
    double argu, fn = 0.0;
    ITEM itm, s;

    printf (" [1;37mPerforming Fourier transform...\n");
    printf ("................................................");
    num_bins = (unsigned long int) floor (sampler.bandwidth * 2.0 / deltaf + 0.1);
    evaluate = (num_bins + 1L) / 2L + (num_bins + 1L) % 2L;
    count = evaluate / 50L;
    count += (count == 0L);
    for (i = 0L; i < evaluate; i++) {
        itm.bin.fn = fn;
        itm.bin.real = 0.0;
        itm.bin.imaginary = 0.0;
        rewind_list (&sdata);
        while (read_list (&sdata, &s)) {
            argu = TWOPI*fn*s.sample.instance;
            itm.bin.real += s.sample.value * cosine (argu);
            itm.bin.imaginary += s.sample.value * sine (argu);
            }
        if (i == 0L)
            itm.bin.imaginary = 0.0;
        if (i == num_bins / 2.0)
            itm.bin.imaginary = 0.0;
        add_to_list (p_ftdata, itm);
        fn += deltaf;
        if (i % count == 0L)
```

```
            printf ("\b");
        }
    printf ("\n");

}

/*
 * perform inverse direct discrete pseudo-random Fourier transform - equation (4.42)
 */

void inv_dprft (sampler, ftdata, sdata, p_idft_data)
SAMPLING_SCHEME sampler;
LIST ftdata, sdata, *p_idft_data;
{

    unsigned long int num_bins, bins_evaluated, even, count, k, n;
    double xa, argu;
    ITEM itm, new_s, s;

    printf ("Performing inverse transform...\n");
    printf ("................................................");
    num_bins = (unsigned long int) floor (sampler.window_width * sampler.bandwidth * 2.0 + 0.1);
    bins_evaluated = (num_bins + 1L) / 2L + (num_bins + 1L) % 2L;
    even = (num_bins + 1L) % 2L;
    count = sampler.num_points / 50L;
    count += (count == 0L);
    rewind_list (&sdata);
    for (k = 0L; k < sampler.num_points; k++) {
        read_list (&sdata, &s);
        rewind_list (&ftdata);
        read_list (&ftdata, &itm);
        xa = itm.bin.real;
        for (n = 1L; n < bins_evaluated - even; n++) {
            read_list (&ftdata, &itm);
            argu = TWOPI*s.sample.instance*itm.bin.fn;
            xa += 2.0 * (itm.bin.real * cosine (argu) + itm.bin.imaginary * sine (argu));
            }
        if (even) {
            read_list (&ftdata, &itm);
            argu = TWOPI*s.sample.instance*itm.bin.fn;
            xa += itm.bin.real * cosine (argu);
            }
        xa /= (double) num_bins;
        new_s.sample.value = xa;
        new_s.sample.instance = s.sample.instance;
        add_to_list (p_idft_data, new_s);
        if (k % count == 0L)
            printf ("\b");
        }
    printf ("\n");

}
```

```
/*****************************************************************************
 *                                                                           *
 * routines to determine exact Fourier coefficients from spectrum estimates *
 *                                                                           *
 *****************************************************************************/

char *append_name (name, extension)
char *name, *extension;
{

    char *new_name, *p_ext;

    new_name = (char *) malloc (MAX_FILENAME_SIZE * sizeof (char));
    new_name = strcpy (new_name, name);
    if ((p_ext = strchr (new_name, '.')) != NULL)
        *p_ext = '\0';
    new_name = strcat (new_name, extension);
    return (new_name);

}

/*
 * create matrix A in file with extension .mat
 */

double matrix_A (sd, bin_step, row, column)
LIST sd;
double bin_step;
int row, column;
{

    double v = 0.0;
    ITEM i;

    rewind_list (&sd);
    if (row == 0 && column == 0) {
        while (read_list (&sd, &i))
            v += 1.0;
        return (v);
        }
    if (row == 0) {
        if (even (column))
            while (read_list (&sd, &i))
                v += sine (TWOPI * (double) (column/2) * bin_step * i.sample.instance);
        else
            while (read_list (&sd, &i))
                v += cosine (TWOPI * (double) ((column + 1)/2) * bin_step * i.sample.instance);
        return (v);
        }
    if (column == 0) {
        if (even (row))
            while (read_list (&sd, &i))
                v += sine (TWOPI * (double) (row/2) * bin_step * i.sample.instance);
        else
            while (read_list (&sd, &i))
                v += cosine (TWOPI * (double) ((row + 1)/2) * bin_step * i.sample.instance);
        return (v);
        }
    if (even (row))
        if (even (column))
```

```
            while (read_list (&sd, &i))
                v += sine (TWOPI * (double) (column/2) * bin_step * i.sample.instance) *
                    sine (TWOPI * (double) (row/2) * bin_step * i.sample.instance);
        else
            while (read_list (&sd, &i))
                v += cosine (TWOPI * (double) ((column + 1)/2) * bin_step * i.sample.instance) *
                    sine (TWOPI * (double) (row/2) * bin_step * i.sample.instance);
    else
        if (even (column))
            while (read_list (&sd, &i))
                v += sine (TWOPI * (double) (column/2) * bin_step * i.sample.instance) *
                    cosine (TWOPI * (double) ((row + 1)/2) * bin_step * i.sample.instance);
        else
            while (read_list (&sd, &i))
                v += cosine (TWOPI * (double) ((column + 1)/2) * bin_step * i.sample.instance) *
                    cosine (TWOPI * (double) ((row + 1)/2) * bin_step * i.sample.instance);
    return (v);

}

int generate_matrix_A (sampler, deltaf, sample_times, filename)
SAMPLING_SCHEME sampler;
double deltaf;
LIST sample_times;
char *filename;
{

    char *matfile_name, key, *invfile_name;
    FILE *matrix_a;
    int num_bins, bins_evaluated, m_size, size, row, column;
    SAMPLING_SCHEME ss;
    double value;

    matfile_name = append_name (filename, ".mat");
    num_bins = (int) floor (sampler.bandwidth * 2.0 / deltaf + 0.1);
    bins_evaluated = (num_bins + 1) / 2 + (num_bins + 1) % 2;
    m_size = 2 * bins_evaluated - 1 - (num_bins + 1) % 2;
    if ((matrix_a = fopen (matfile_name, "rb")) != NULL) {
        rewind (matrix_a);
        if (fread ((int *) &size, sizeof (int), 1, matrix_a) &&
            fread ((SAMPLING_SCHEME *) &ss, sizeof (SAMPLING_SCHEME), 1, matrix_a))
            if (size == m_size && equal_schemes (ss, sampler)) {
                printf ("\n [1;37mRequired %dx%d matrix already exists in file %s\n", m_size,
m_size, matfile_name);
                fclose (matrix_a);
                return (1);
                }
            else {
                printf (" [0;37m%s exists. Over-write (y/n)?  [1;33m", matfile_name);
                while ((key = getch()) != 'y' && key != 'n')
                    ;
                printf ("%c\n", key);
                if (key == 'n') {
                    fclose (matrix_a);
                    return (0);
                    }
                else { /* ensure inverse matrix file also over-written */
                    invfile_name = append_name (filename, ".inv");
                    if (remove (invfile_name) == -1 && errno != ENOENT) error (FILE_ERR);
                    free ((char *) invfile_name);
```

```
                    }
                }
            }
        printf ("\n [1;37m");
        if ((matrix_a = fopen (matfile_name, "wb")) == NULL) error (FILE_ERR);
        printf ("Creating %dx%d matrix in file %s...", m_size, m_size, matfile_name);
        printf (" [0;37m rows made ( [s0)");
        fwrite ((int *) &m_size, sizeof (int), 1, matrix_a);
        fwrite ((SAMPLING_SCHEME *) &sampler, sizeof (SAMPLING_SCHEME), 1, matrix_a);
        for (row = 0; row < m_size; row++) {
            for (column = 0; column < m_size; column++) {
                value = matrix_A (sample_times, deltaf, row, column);
                fwrite ((double *) &value, sizeof (double), 1, matrix_a);
                }
            printf (" [u%d)", row + 1);
            }
        printf ("\n");
        free ((char *) matfile_name);
        fclose (matrix_a);
        return (1);

}

/*                                -1
 * calculate inverse of matrix, A   and place in file with extension .inv
 */

void copy_matrix_file (source_name, dest_name)
char *source_name, *dest_name;
{

    FILE *source, *dest;
    int size;
    SAMPLING_SCHEME sampler;
    double value;

    if ((source = fopen (source_name, "rb")) == NULL) error (FILE_ERR);
    if ((dest = fopen (dest_name, "wb")) == NULL) error (FILE_ERR);
    rewind (source);
    fread ((int *) &size, sizeof (int), 1, source);
    fread ((SAMPLING_SCHEME *) &sampler, sizeof (SAMPLING_SCHEME), 1, source);
    while (fread ((double *) &value, sizeof (double), 1, source) != 0)
        fwrite ((double *) &value, sizeof (double), 1, dest);
    fclose (source);
    fclose (dest);

}

ludcmp (in_file, n, indx, p_d)
FILE *in_file;
int n, *indx;
short *p_d;
{

    int i, j, k, i_max;
    double max_a, sum, x, y, dum;
    double *scale_v;

    scale_v = (double *) malloc ((n + 1) * sizeof (double));
    *p_d = 1;  /* no row interchanges yet */
```

```
/* loop over rows to get the implicit scaling information */
rewind (in_file);
for (i = 1; i <= n; i++) {
    max_a = 0.0;
    for (j = 1; j <= n; j++) {
        fread ((double *) &x, sizeof (double), 1, in_file);
        if (fabs (x) > max_a)
            max_a = fabs (x);
        }
    if (max_a == 0.0) error (SINGULAR_MATRIX);
    scale_v[i] = 1.0/max_a;  /* save the scaling */
    }
/* loop over columns of Crout's method */
for (j = 1; j <= n; j++) {
    for (i = 1; i <= j - 1; i++) {
        fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
        fread ((double *) &sum, sizeof (double), 1, in_file);
        for (k = 1; k <= i - 1; k++) {
            fseek (in_file, (long) (((i - 1)*n + (k - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &x, sizeof (double), 1, in_file);
            fseek (in_file, (long) (((k - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &y, sizeof (double), 1, in_file);
            sum -= x*y;
            }
        fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
        fwrite ((double *) &sum, sizeof (double), 1, in_file);
        }
    max_a = 0.0;  /* initialise for the search for largest pivot element */
    for (i = j; i <= n; i++) {
        fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
        fread ((double *) &sum, sizeof (double), 1, in_file);
        for (k = 1; k <= j - 1; k++) {
            fseek (in_file, (long) (((i - 1)*n + (k - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &x, sizeof (double), 1, in_file);
            fseek (in_file, (long) (((k - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &y, sizeof (double), 1, in_file);
            sum -= x*y;
            }
        fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
        fwrite ((double *) &sum, sizeof (double), 1, in_file);
        dum = scale_v[i]*fabs (sum);  /* figure of merit for the pivot */
        if (dum >= max_a) {  /* is it better than the best so far? */
            i_max = i;
            max_a = dum;
            }
        }
    if (j != i_max) {
        for (k = 1; k <= n; k++) {  /* interchange rows */
            fseek (in_file, (long) (((i_max - 1)*n + (k - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &dum, sizeof (double), 1, in_file);
            fseek (in_file, (long) (((j - 1)*n + (k - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &x, sizeof (double), 1, in_file);
            fseek (in_file, (long) (- sizeof (double)), SEEK_CUR);
            fwrite ((double *) &dum, sizeof (double), 1, in_file);
            fseek (in_file, (long) (((i_max - 1)*n + (k - 1))*sizeof (double)), SEEK_SET);
            fwrite ((double *) &x, sizeof (double), 1, in_file);
            }
        *p_d = -*p_d;  /* change the parity of *p_d */
        scale_v[i_max] = scale_v[j];  /* interchange the scale factor */
        }
```

```
        indx[j] = i_max;
        fseek (in_file, (long) (((j - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
        fread ((double *) &dum, sizeof (double), 1, in_file);
        if (dum == 0.0) {
            dum = DBL_MIN;
            fseek (in_file, (long) (- sizeof (double)), SEEK_CUR);
            fwrite ((double *) &dum, sizeof (double), 1, in_file);
            }
        if (j != n) {
            fseek (in_file, (long) (((j - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &dum, sizeof (double), 1, in_file);
            for (i = j + 1; i <= n; i++) {
                fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
                fread ((double *) &x, sizeof (double), 1, in_file);
                if (dum == DBL_MIN)
                    x = HUGE_VAL;
                else
                    x /= dum;
                fseek (in_file, (long) (- sizeof (double)), SEEK_CUR);
                fwrite ((double *) &x, sizeof (double), 1, in_file);
                }
            }
        printf (" [u%d)", j);
        }  /* go back for the next column in the reduction */
    free ((double *) scale_v);
    return;

}

lubksb (in_file, n, indx, b)
FILE *in_file;
int n, *indx;
double *b;
{
    int ii = 0, i, ll, j;
    double sum, x;

    for (i = 1; i <= n; i++) {
        ll = indx[i];
        sum = b[ll];
        b[ll] = b[i];
        if (ii != 0)
            for (j = ii; j <= i - 1; j++) {
                fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
                fread ((double *) &x, sizeof (double), 1, in_file);
                sum -= x*b[j];
                }
        else
            if (sum != 0.0)
                ii = i;
        b[i] = sum;
        }
    for (i = n; i >= 1; i--) {
        sum = b[i];
        if (i < n)
            for (j = i + 1; j <= n; j++) {
                fseek (in_file, (long) (((i - 1)*n + (j - 1))*sizeof (double)), SEEK_SET);
                fread ((double *) &x, sizeof (double), 1, in_file);
                sum -= x*b[j];
                }
```

```
        fseek (in_file, (long) (((i - 1)*n + (i - 1))*sizeof (double)), SEEK_SET);
        fread ((double *) &x, sizeof (double), 1, in_file);
        if (x == HUGE_VAL)
            b[i] = 0.0;
        else
            b[i] = sum/x;
        }
    return;

}

void inverse_matrix (filename)
char *filename;
{

    FILE *matrix, *work, *inverse;
    int i, j, *indx, size;
    double x, *y, *b;
    short d;
    char *tempfile_name, *matfile_name, *invfile_name;

    printf (" [1;37m");
    matfile_name = append_name (filename, ".mat");
    invfile_name = append_name (filename, ".inv");
    if ((inverse = fopen (invfile_name, "rb")) != NULL) {
        fclose (inverse);
        printf ("Matrix inverse already exists in file %s\n", invfile_name);
        return;
        }
    if ((matrix = fopen (matfile_name, "rb")) == NULL) error (FILE_ERR);
    if ((inverse = fopen (invfile_name, "w+b")) == NULL) error (FILE_ERR);
    if ((tempfile_name = tempnam ("c:\tmp", "mat")) == NULL) error (NO_TEMP);
    copy_matrix_file (matfile_name, tempfile_name);
    if ((work = fopen (tempfile_name, "r+b")) == NULL) error (FILE_ERR);
    rewind (matrix);
    fread ((int *) &size, sizeof (int), 1, matrix);
    printf ("Creating inverse of matrix... [0;37m columns processed ( [s0)");
    /* set up identity matrix */
    for (i = 1; i <= size; i++)
        for (j = 1; j <= size; j++) {
            if (i == j)
                x = 1.0;
            else
                x = 0.0;
            fwrite ((double *) &x, sizeof (double), 1, inverse);
            }
    /* LU decompose the matrix just once */
    indx = (int *) malloc ((size + 1) * sizeof (int));
    ludcmp (work, size, indx, &d);
    /* find inverse by columns */
    y = (double *) malloc ((size + 1) * sizeof (double));
    b = (double *) malloc ((size + 1) * sizeof (double));
    for (j = 1; j <= size; j++) {
        for (i = 1; i <= size; i++) {
            fseek (inverse, (long) (((i - 1)*size + (j - 1))*sizeof (double)), SEEK_SET);
            fread ((double *) &y[i], sizeof (double), 1, inverse);
            b[i] = (i == j) ? 1.0 : 0.0;
            }
        lubksb (work, size, indx, y);
        for (i = 1; i <= size; i++) {
```

```
            fseek (inverse, (long) (((i - 1)*size + (j - 1))*sizeof (double)), SEEK_SET);
            fwrite ((double *) &y[i], sizeof (double), 1, inverse);
            }
        printf (" [u%d]  ", j);
        }
    printf ("\n");
    free ((int *) indx);
    free ((double *) y);
    free ((char *) matfile_name);
    free ((char *) invfile_name);
    fclose (matrix);
    fclose (work);
    fclose (inverse);
    if (remove (tempfile_name) == -1) error (FILE_ERR);

}

/*
 * determine exact Fourier coefficients using inverse matrix and estimated coeffs
 */

void calc_coefficients (filename, sampler, deltaf, ftdata, p_newdata)
char *filename;
SAMPLING_SCHEME sampler;
double deltaf;
LIST ftdata, *p_newdata;
{


    char *invfile_name;
    FILE *inverse;
    int num_bins, bins_evaluated, i;
    ITEM estimate, exact;
    double fn = 0.0, sum, in_val;
    enum {RE = 0, IM = 1, DONE = 2} state;

    invfile_name = append_name (filename, ".inv");
    if ((inverse = fopen (invfile_name, "rb")) == NULL) error (FILE_ERR);
    rewind (inverse);
    num_bins = (int) floor (sampler.bandwidth * 2.0 / deltaf + 0.1);
    bins_evaluated = (num_bins + 1) / 2 + (num_bins + 1) % 2;
    for (i = 0; i < bins_evaluated; i++) {
        exact.bin.fn = fn;
        for (state = RE; state != DONE; state++) {
            sum = 0.0;
            rewind_list (&ftdata);
            while (read_list (&ftdata, &estimate)) {
                if (fread ((double *) &in_val, sizeof (double), 1, inverse) == 0) error (EOF_FOUND);
                sum += in_val*estimate.bin.real;
                if (!(estimate.bin.fn == 0.0 ||
                      (estimate.bin.fn == sampler.bandwidth && even (num_bins)))) {
                    if (fread ((double *) &in_val, sizeof (double), 1, inverse) == 0)
                        error (EOF_FOUND);
                    sum += in_val*estimate.bin.imaginary;
                    }
                }
            if (state == RE) {
                exact.bin.real = sum;
                if (i == 0) {
                    state = IM;
```

```
                    exact.bin.imaginary = 0.0;
                    }
                if (i == num_bins / 2.0) {
                    state = IM;
                    exact.bin.imaginary = 0.0;
                    }
                }
            else
                exact.bin.imaginary = sum;
            }
        add_to_list (p_newdata, exact);
        fn += deltaf;
        }
    fclose (inverse);

}

/*****************************************************************************
 *                                                                           *
 * user interface routines                                                   *
 *                                                                           *
 * void scanf_double (double *p_var);                                        *
 * void display_title (void)                                                 *
 * void enter_sampler_parameters (SAMPLING_SCHEME *p_sampler);               *
 * void enter_signal_data (LIST *p_input);                                   *
 * void plot_analysis (LIST in_freqs, LIST fourier_coeffs, SAMPLING_SCHEME sampler); *
 * void plot_errors (SAMPLING_SCHEME sampler, LIST s_data, LIST inv_data);   *
 *                                                                           *
 *****************************************************************************/

void scanf_double (p_var)
double *p_var;
{

    char string[30], *denominator;

    scanf (" \t\n%s", string);
    *p_var = strtod (string, &denominator);
    if (*denominator == '/')
        *p_var /= atof (++denominator);

}

void display_title (void)
{

    printf (" [1;37m [2JSub-Nyquist Sampling Techniques\n");
    printf ("P C Bagshaw    July 1990\n");
    printf ("Pseudo-random Discrete Fourier Transform Simulator\n");

}

void enter_sampler_parameters (p_sampler)
SAMPLING_SCHEME *p_sampler;
{

    char key;
    int zero_in_list = 0;
    ITEM itm;
```

```
    printf (" [1;37m\nSampling Scheme Parameters.\n");
    printf (" [0;37mNumber of sampling points (< %d):  [1;33m", INT_MAX);
    scanf ("%d", &(p_sampler->num_points));
    printf (" [0;37mModel (U)niform sampling, (A)dditive pseudo-random sampling,\n");
    printf ("or periodic sampling with (D)ither? (u/a/d):  [1;33m");
    /* assume machine uses ASCII character set */
    while ((key = (int) getch() | 32) != 'u' && key != 'a' && key != 'd')
        ;
    printf ("%c\n", key);
    p_sampler->scheme = key;
    p_sampler->num_fs = 0;
    initialise_list (&p_sampler->fs, LIST_SAMPLE_FREQS);
    switch (p_sampler->scheme) {
        case UNIFORM:
            printf (" [0;37mEnter uniform sampling frequency:  [1;33m");
            scanf_double (&itm.frequency);
            add_to_list (&p_sampler->fs, itm);
            p_sampler->num_fs = 1;
            break;
        case ADDITIVE:
            printf (" [0;37mEnter pseudo-random sampling frequencies (end with zero):  [1;33m");
            scanf_double (&itm.frequency);
            while (itm.frequency > 0.0) {
                add_to_list (&p_sampler->fs, itm);
                p_sampler->num_fs++;
                scanf_double (&itm.frequency);
                }
            break;
        case DITHER:
            printf (" [0;37mEnter periodic sampling frequency:  [1;33m");
            scanf_double (&itm.frequency);
            add_to_list (&p_sampler->fs, itm);
            printf (" [0;37mEnter possible positive dither in terms of fractions, 0   x < 1/2 ");
            printf ("of\nthe sampling period, %f (end with x out of range):  [1;33m",
1.0/itm.frequency);
            scanf_double (&itm.frequency);
            while (itm.frequency >= 0.0 && itm.frequency < 0.5) {
                if (itm.frequency == 0.0)
                    zero_in_list = 1;
                else
                    add_to_list (&p_sampler->fs, itm);
                p_sampler->num_fs += 2;
                scanf_double (&itm.frequency);
                }
            p_sampler->num_fs -= zero_in_list;
            break;
        default:
            break;
        }
    if (p_sampler->num_fs == 0) error (NO_SF);
    if (p_sampler->num_fs == 1)
        p_sampler->scheme = UNIFORM;
    printf (" [0;37m");

}
```

```
void enter_signal_data (p_input)
LIST *p_input;
{

    ITEM itm;

    printf (" [1;37m\nInput Signal Details.\n");
    printf (" [0;37mEnter frequency, amplitude and phase of the input signal harmonics.\n");
    printf ("(terminate with a negative frequency)\n");
    itm.element.frequency = 0.0;
    while (itm.element.frequency >= 0.0) {
        printf (" [0;37m* freq:  [1;33m");
        scanf_double (&itm.element.frequency);
        if (itm.element.frequency >= 0.0) {
            printf (" [0;37m* amp:  [1;33m");
            scanf_double (&itm.element.amplitude);
            if (itm.element.frequency == 0.0)
                itm.element.phase = 0.0;
            else {
                printf (" [0;37m* phase (degrees):  [1;33m");
                scanf_double (&itm.element.phase);
                itm.element.phase *= PI / 180.0;
                }
            add_to_list (p_input, itm);
            }
        }
    rewind_list (p_input);
    if (!read_list (p_input, &itm)) error (NO_INPUT);
    printf (" [0;37m");

}

/*
 * display frequency domain information
 */

typedef enum {ESTIMATE, EXACT} COEFF_TYPE;

void plot_analysis (in_freqs, sampler, fourier_coeffs, amp_control)
LIST in_freqs, fourier_coeffs;
SAMPLING_SCHEME sampler;
COEFF_TYPE amp_control;
{

    struct videoconfig vc;
    double max = 0.1, x_scale, amp_scale, phase_scale;
    double amplitude, phase;
    char key;
    ITEM itm;

    if (!set_mode()) error (GRAPHICS_MODE);
    _getvideoconfig (&vc);

    /* print labels */
    printf (" [1;37m");
    _settextposition (1, 1);
    if (amp_control == ESTIMATE)
        _outtext ("Estimate Signal Analysis using Pseudo-random sampling and Fourier transform");
    else
        _outtext ("Exact Signal Analysis using Transform and Inverse Matrix");
```

```
_settextposition (20, 1);
_outtext ("Phase\n '");
_settextposition (28, 1);
_outtext ("-'");
_settextposition (19, 80);
_outtext ("f");

/* plot axes */
_setcolor (15);             /* white */
_setlogorg (0, (short) vc.numypixels*0.59);
x_scale = (double) (vc.numxpixels-1)/sampler.bandwidth;
rewind_list (&fourier_coeffs);
while (read_list (&fourier_coeffs, &itm)) {
    convert_complex (itm.bin.real, itm.bin.imaginary, &amplitude, &phase);
    if (amp_control != EXACT) {
        amplitude /= (itm.bin.fn == 0.0) + 1.0;
        amplitude *= 2.0/sampler.num_points;
        }
    if (amplitude > max)
        max = amplitude;
    }
amp_scale = (double) (-vc.numypixels) * 0.525 / max;
_moveto (0,(short) (-vc.numypixels * 0.525));
_lineto (0, 0);
_lineto (vc.numxpixels, 0);
_setlogorg (0, (short) (vc.numypixels*0.795));
phase_scale = (double) (-vc.numypixels)/PI*0.118;
_moveto (0, (short) phase_scale*PI);
_lineto (0, (short) -phase_scale*PI);
_moveto (0, 0);
_lineto (vc.numxpixels, 0);


/* plot amplitude and phase information of transform output */
_setcolor (14);             /* yellow */
rewind_list (&fourier_coeffs);
while (read_list (&fourier_coeffs, &itm)) {
    convert_complex (itm.bin.real, itm.bin.imaginary, &amplitude, &phase);
    if (amp_control != EXACT) {
        amplitude /= (itm.bin.fn == 0.0) + 1.0;
        amplitude *= 2.0/sampler.num_points;
        }
    _setlogorg (0, (short) vc.numypixels*0.59);
    _moveto ((short) (itm.bin.fn*x_scale), 0);
    _lineto ((short) (itm.bin.fn*x_scale), (short) (amplitude*amp_scale));
    _setlogorg (0, (short) (vc.numypixels*0.795));
    _moveto ((short) (itm.bin.fn*x_scale), 0);
    _lineto ((short) (itm.bin.fn*x_scale), (short) (phase*phase_scale));
    }

/* plot sample frequencies */
_setlogorg (0, (short) vc.numypixels*0.59);
_setcolor (13);                 /* light magenta */
rewind_list (&sampler.fs);
while (read_list (&sampler.fs, &itm)) {
    _moveto ((short) (itm.frequency*x_scale), 0);
    _lineto ((short) (itm.frequency*x_scale), 10);
    }
/* plot input frequencies */
_setcolor (10);                 /* light green */
```

```
    rewind_list (&in_freqs);
    while (read_list (&in_freqs, &itm)) {
        _moveto ((short) (itm.element.frequency*x_scale), 0);
        _lineto ((short) (itm.element.frequency*x_scale), 7);
        }

    /* highlight successive bins and display details of highlighted bin */
    key = -1;
    rewind_list (&fourier_coeffs);
    while (key != 'e' && read_list (&fourier_coeffs, &itm)) {
        convert_complex (itm.bin.real, itm.bin.imaginary, &amplitude, &phase);
        if (amp_control != EXACT) {
            amplitude /= (itm.bin.fn == 0.0) + 1.0;
            amplitude *= 2.0/sampler.num_points;
            }
        _setcolor (12);             /* red */
        _setlogorg (0, (short) vc.numypixels*0.59);
        _moveto ((short) (itm.bin.fn*x_scale), 0);
        _lineto ((short) (itm.bin.fn*x_scale), (short) (amplitude*amp_scale));
        _setlogorg (0, (short) (vc.numypixels*0.795));
        _moveto ((short) (itm.bin.fn*x_scale), 0);
        _lineto ((short) (itm.bin.fn*x_scale), (short) (phase*phase_scale));
        _settextposition (2, 1);
        printf ("frequency: %.4f ", itm.bin.fn);
        _settextposition (2, 25);
        printf ("amplitude: %.4f ", amplitude);
        _settextposition (2, 50);
        printf ("phase (degrees): %7.4f   ", phase * 180.0 / PI);
        key = getch ();
        _setcolor (14);             /* yellow */
        _setlogorg (0, (short) vc.numypixels*0.59);
        _moveto ((short) (itm.bin.fn*x_scale), 0);
        _lineto ((short) (itm.bin.fn*x_scale), (short) (amplitude*amp_scale));
        _setlogorg (0, (short) (vc.numypixels*0.795));
        _moveto ((short) (itm.bin.fn*x_scale), 0);
        _lineto ((short) (itm.bin.fn*x_scale), (short) (phase*phase_scale));
        }
    _setvideomode (_DEFAULTMODE);
    return;

}

/*
 * display time domain information
 */

void plot_errors (sampler, s_data, inv_data)
SAMPLING_SCHEME sampler;
LIST s_data, inv_data;
{

    struct videoconfig vc;
    int k;
    double x_scale, y_scale, err, max_e = 0.0, min_e = HUGE;
    ITEM s, i;

    if (!set_mode()) error (GRAPHICS_MODE);
    _getvideoconfig (&vc);
    rewind_list (&s_data);
    rewind_list (&inv_data);
```

```
    while (read_list (&s_data, &s) && read_list (&inv_data, &i)) {
        err = fabs (i.sample.value - s.sample.value);
        max_e = (err > max_e) ? err : max_e;
        min_e = (err < min_e) ? err : min_e;
        }
    x_scale = (double) (vc.numxpixels-1)/sampler.num_points;
    y_scale = (double) (-vc.numypixels) / max_e * 0.7;
    _setcolor (15);             /* white */
    _setlogorg (0, (short) vc.numypixels*0.8);
    _moveto (0,(short) (-vc.numypixels*0.7));
    _lineto (0, 0);
    _lineto (vc.numxpixels, 0);
    _setcolor (14);             /* yellow */
    printf (" [1;37mError in calculating sample values through inverse transform\n\n");
    printf ("Maximum error: %e\tMinimum error: %e\n", max_e, min_e);
    _settextposition (26, 1);
    printf ("0");
    _settextposition (26, 35);
    printf ("Sample");
    _settextposition (26, 77);
    printf ("%d", sampler.num_points - 1);
    rewind_list (&s_data);
    rewind_list (&inv_data);
    for (k = 0; k < sampler.num_points; k++) {
        read_list (&s_data, &s);
        read_list (&inv_data, &i);
        err = fabs (i.sample.value - s.sample.value);
        if (k == 0)
            _moveto (0, (short) (err*y_scale));
        _lineto ((short) (k*x_scale), (short) (err*y_scale));
        }
    getch ();
    _setvideomode (_DEFAULTMODE);

}

int set_mode ()
{

    if (_setvideomode (_VRES16COLOR))
        return (_VRES16COLOR);
    if (_setvideomode (_ERESCOLOR))
        return (_ERESCOLOR);
    if (_setvideomode (_MRES16COLOR))
        return (_MRES16COLOR);
    else
        return (0);

}
```

```
/****************
 *              *
 * main program *
 *              *
 ****************/

SAMPLING_SCHEME sampler;
LIST input_signal, sample_data, ft_data, idft_data, exact_ftdata;
double bin_step = 0.0, old_step;
char key = '0', matrix_name[MAX_FILENAME_SIZE];
int need_sampler_parameters = 1;
int need_signal_data = 1;
int need_generate_data = 1;
int need_prdft_calculated = 1;
int need_invprdft_calculated = 1;

void satisfy_input_needs ()
{

    if (need_sampler_parameters) {
        enter_sampler_parameters (&sampler);
        calc_bandwidth (&sampler);
        printf (" [1;37m\nSystem Bandwidth: %.3f\n", sampler.bandwidth);
        need_sampler_parameters = 0;
        free_list (&sample_data);
        need_generate_data = 1;
        }
    if (need_signal_data) {
        enter_signal_data (&input_signal);
        need_signal_data = 0;
        free_list (&sample_data);
        need_generate_data = 1;
        }
    if (need_generate_data) {
        printf (" [1;37m\nGenerating sampling data...\n");
        random_sampler (&sampler, input_signal, &sample_data);
        need_generate_data = 0;
        free_list (&ft_data);
        need_prdft_calculated = 1;
        }
}

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

    initialise_list (&input_signal, FREQ_LIST);
    initialise_list (&sample_data, SAMPLE_DATA);
    initialise_list (&ft_data, FOURIER_TRANSFORM_DATA);
    initialise_list (&idft_data, SAMPLE_DATA);
    initialise_list (&exact_ftdata, FOURIER_TRANSFORM_DATA);
    display_title ();
    satisfy_input_needs ();
    while (key != '6') {
        printf (" [1;37m [2J\nSimulation Options. [0;37m\n");
        printf ("1. Change sampler parameters\n");
        printf ("2. Change input signal description\n");
        printf ("3. Calculate pseudo-random discrete Fourier transform only and display spectrum\n");
        printf ("4. Evaluate PRDFT, the Inverse PRDFT and display errors and spectrum\n");
```

```
printf ("5. Calculate PRDFT, determine exact coefficients from it and display spectra\n");
printf ("6. Quit\n");
printf ("Enter choice (1-6):  [1;33m");
while ((key = getch()) < '1' || key > '6')
    ;
printf ("%c\n", key);
if (key == '1') {
    free_list (&sampler.fs);
    need_sampler_parameters = 1;
    }
if (key == '2') {
    free_list (&input_signal);
    need_signal_data = 1;
    }
satisfy_input_needs ();
if (key == '3' || key == '5') {
    printf (" [0;37mEnter bin step (  %f):  [1;33m", 1.0/sampler.window_width);
    old_step = bin_step;
    scanf_double (&bin_step);
    if (bin_step != old_step) {
        free_list (&ft_data);
        need_prdft_calculated = 1;
        }
    }
if (key == '4') {
    old_step = bin_step;
    bin_step = 1.0/sampler.window_width;
    if (bin_step != old_step) {
        free_list (&ft_data);
        need_prdft_calculated = 1;
        }
    }
if ((key == '3' || key == '4') && need_prdft_calculated) {
    printf ("\n");
    dprft (sampler, bin_step, sample_data, &ft_data);
    need_prdft_calculated = 0;
    }
switch (key) {
    case '3':
        plot_analysis (input_signal, sampler, ft_data, ESTIMATE);
        break;
    case '4':
        if (need_invprdft_calculated) {
            inv_dprft (sampler, ft_data, sample_data, &idft_data);
            need_invprdft_calculated = 0;
            }
        plot_errors (sampler, sample_data, idft_data);
        plot_analysis (input_signal, sampler, ft_data, ESTIMATE);
        break;
    case '5':
        printf (" [0;37mEnter name of matrix (filename without extension):  [1;33m");
        scanf (" \t\n%s", matrix_name);
        while (!generate_matrix_A (sampler, bin_step, sample_data, matrix_name)) {
            printf (" [0;37mEnter name of matrix (filename without extension):  [1;33m");
            scanf (" \t\n%s", matrix_name);
            }
        inverse_matrix (matrix_name);
        if (need_prdft_calculated) {
            dprft (sampler, bin_step, sample_data, &ft_data);
            need_prdft_calculated = 0;
```

```
                }
            calc_coefficients (matrix_name, sampler, bin_step, ft_data, &exact_ftdata);
            plot_analysis (input_signal, sampler, ft_data, ESTIMATE);
            plot_analysis (input_signal, sampler, exact_ftdata, EXACT);
            free_list (&exact_ftdata);
            break;
        default:
            break;
        }
    }
exit (0);

}
```

APPENDIX D

Source Code of DFT and Inverse DFT using NAG Library Routines.

```
program nag_dft (input, output);

const   max_fft = 2048;
        pi = 3.14159265358979323849;

type    fft_data = array [0..max_fft - 1] of real;
        time_data = array [0..max_fft] of real;

var     n, ifail : integer;
        fs : real;
        xa, xx, work, f_nag, x_nag, f_pra, x_pra : fft_data;
        tk : time_data;

procedure C06FAF (var x:fft_data; const n:integer; var work:fft_data;
                  var ifail:integer);fortran77;
procedure C06GBF (var x:fft_data; const n:integer;
                  var ifail:integer);fortran77;
procedure C06FBF (var x:fft_data; const n:integer; var work:fft_data;
                  var ifail:integer);fortran77;

procedure GENERATE_DATA (var num_samples:integer; var sample_freq:real;
                         var x:fft_data; var t:time_data);
var     j : integer;
        scale : real;
begin {generate_data}
        readln (num_samples);
        readln (sample_freq);
        scale := 0.0;
        for j := 0 to num_samples - 1 do
            begin
            t[j] := scale / sample_freq;
            x[j] := cos (2.0 * pi * 4.0 * t[j]);
            scale := scale + 1.0
            end;
        t[n] := scale / sample_freq
end; {generate_data}

procedure DPRFT (var x:fft_data; const n:integer; const bandwidth:real;
                 const t:time_data; var work:fft_data);
var     fn, argu, re, im : real;
        i, j, num_bins, evaluate : integer;
begin {dprft}
        fn := 0.0;
        num_bins := round (t[n] * 2.0 * bandwidth);
        evaluate := (num_bins + 1) div 2 + (num_bins + 1) mod 2;
        for i := 0 to evaluate - 1 do
            begin
            re := 0.0;
            im := 0.0;
            for j := 0 to n - 1 do
                begin
                argu := 2.0 * pi * fn * t[j];
                re := re + x[j] * cos (argu);
                im := im + x[j] * sin (argu)
                end;
            im := -im;
            if i = 0 then
                work[0] := re
            else
                begin
```

```
                work[num_bins - i] := im;
                work[i] := re
                end;
            fn := fn + 1.0/t[n]
            end;
        x := work
end; {dprft}

procedure INVERSE_DPRFT (var ftd:fft_data; const n:integer;
                         const bandwidth:real; const t:time_data;
                         var work:fft_data);
var     fn, argu : real;
        i, j, num_bins, evaluate : integer;
begin {inverse_dprft}
        num_bins := round (t[n] * 2.0 * bandwidth);
        evaluate := (num_bins + 1) div 2 + (num_bins + 1) mod 2;
        for j := 0 to n - 1 do
            begin
            fn := 0.0;
            work[j] := ftd[0];
            for i := 1 to evaluate - 1 - (num_bins + 1) mod 2 do
                begin
                fn := fn + 1.0/t[n];
                argu := 2.0 * pi * t[j] * fn;
                work[j] := work[j] + 2.0 * (ftd[i] * cos (argu) -
                                            ftd[num_bins - i] * sin (argu))
                end;
            if (num_bins mod 2 = 0) then
                begin
                fn := fn + 1.0/t[n];
                work[j] := work[j] +
                           ftd[num_bins div 2] * cos (2.0 * pi * t[j] * fn)
                end;
            work[j] := work[j] / num_bins
            end;
        ftd := work
end; {inverse_dprft}

procedure DISPLAY_T_DOMAIN (const xa, x_nag, x_pra:fft_data; const n:integer);
var     j : integer;
        errorx, max_nag_err, max_pra_err: real;
begin {display_t_domain}
        max_nag_err := 0.0;
        max_pra_err := 0.0;
        writeln ('Input sequence as restored by IDPRFT');
        for j := 0 to n - 1 do
            begin
            errorx := abs (xa[j] - x_nag[j]);
            if errorx > max_nag_err then
                max_nag_err := errorx;
            errorx := abs (xa[j] - x_pra[j]);
            if errorx > max_pra_err then
                max_pra_err := errorx;
            writeln (j, xa[j], x_pra[j], errorx)
            end;
        writeln ('Maximum error in restoring data by NAG: ', max_nag_err);
        writeln ('Maximum error in restoring data by IDPRFT:', max_pra_err)
end; {display_t_domain}
```

```
begin {main program}
        GENERATE_DATA (n, fs, xa, tk);
                {perform DFT using NAG library routine}
        xx := xa;
        ifail := 0;
        C06FAF (xx, n, work, ifail);
        f_nag := xx;
                {perform IDFT using NAG library routines}
        C06GBF (xx, n, ifail);
        C06FBF (xx, n, work, ifail);
        x_nag := xx;
                {perform DFT using PSEUDO-RANDOM algorithm}
        xx := xa;
        DPRFT (xx, n, fs/2.0, tk, work);
        f_pra := xx;
                {perform IDFT using PSEUDO-RANDOM algorithm}
        INVERSE_DPRFT (xx, n, fs/2.0, tk, work);
        x_pra := xx;
                {output information to user}
        DISPLAY_T_DOMAIN (xa, x_nag, x_pra, n)
end. {main program}
```

BIBLIOGRAPHY

Baranov, L. A.

"Error estimates of the restoration of a continuous random signal when sampling is irregular."

Transactions in Telecommunications and Radio Engineering.

Vol. 38, No.8, August 1983, pp37-39

Bilinsky, I. Ya; Borovik, Yu. F. & Mikelson, A. K.

"Complexity-reduced discrete Fourier transform." in

'Signal Processing II: Theories and Applications.'

Schüssler, H. W. (editors)

Proc. EUSIPCO-83: Second European Signal Processing Conference. pp743-746

Bilinsky, I. Ya. & Mikelson, A. K.

СТОХАСТИЧЕСКАЯ ЦИФРОВАЯ ОБРАБТКА НЕПРЕРЫВНЫХ СИГНАЛОВ

('Stochastic Digital Sampling of Continuous Signals.')

Zinatne, Riga, Latvia, 1983 (in Russian)

Bilinsky. I. Ya; Mikelson, A. K. & Yakubaitis, S.

"Method for reducing the variance of a restored randomly sampled signal."

Institute of Electronics and Computational Technology of the Latvian Academy of Science. 1985, Latv. PSR Zinat. Akad. Vestis Fiz. Teh. Zinat. Ser. (USSR)

No.5, pp106-115

(CODEN: LZFTA6, ISSN: 0321-1673, In Russian.)

Bilinsky, I. Ya; Nemirovsky, R. F. & Strautmanis, G. F.

"Wideband signal processing by general-purpose signal processors."

Proc. Seventh European Conference on Circuit Theory and Design.

Prague, Czechoslovakia, 2-6 Sept. 1985, pp371-374

Blahut, R. E.

'Fast Algorithms for Digital Signal Processing.'

Addison-Wiley, Reading, Massachusetts, 1985

Gold, B. & Rader, C. M.

'Digital Processing of Signals.'

McGraw-Hill, New York, 1969

Kernighan, B. W. & Ritchie, D. M.

'The C Programming Language.'

Second Edition. Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1988

Koffman, E. B.

'Problem Solving and Structured Programming in Pascal.'

Second Edition. Addison-Wesley, 1985

Leneman, O. A. Z.

"Random sampling of random processes: Impulse processes."

Information and Control, Vol. 9, 1966, pp347-363

Masry, E.

"Poisson sampling and spectral estimation of continuous-time processes."

IEEE Transactions on Information Theory, Vol. IT-24, No.2, March 1978, pp173-183

Masry, E; Klamer, D. & Mirabile, C.

"Spectral estimation of continuous-time processes: Performance comparison between periodic and Poisson sampling schemes."

IEEE Transactions on Automatic Control, Vol. AC-23, No.4, August 1978, pp679-685

Masry, E. & Lui, M. C.

"Discrete-time spectral estimation of continuous-parameter processes: A new consistent estimate."

IEEE Transactions on Information Theory, Vol. IT-22, No.3, May 1976, pp298-312

Wold, E. H. & Dippé, M. A. Z.

"Alias-free sound synthesis by stochastic sampling."

Proc. 1985 International Computer Music Conference, Vancouver. pp39-46

REFERENCES

[1] Rossi, J. P.

"Sub-Nyquist-encoded PCM NTSC color television."

SMPTE Journal, Vol. 85, No.1, January 1976, pp1-6

[2] Führen, M. & Den Dulk, R. C.

"A new despreading method based on sub-Nyquist sampling." in

'Signal Processing III: Theories and Applications.'

Young, I. T. et al. (editors)

Proc. EUSIPCO-86: Third European Signal Processing Conference.

The Hague, Netherlands, 2-5 Sept. 1986, Vol. 1, pp49-52

[3] Dunlop, J. & Smith, D. G.

'Telecommunications Engineering.'

Van Nostrand Reinhold (UK), Wokingham, 1984

[4] Benjamin, R.

"Orthogonally aliased Fourier transforms for the analysis of sparsely populated frequency spectra."

Proc IEE, Vol. 124, No.6, June 1977, pp508-510

[5] Brigham, E. Oran.

'The Fast Fourier Transform and Its Applications.'

Prentice-Hall International, Englewood Cliffs, New Jersey, 1988

[6] Underhill, M. J; Sarhadi, M. & Aitchison, C. S.

"Fast-sampling frequency meter."

Electronics Letters, Vol. 14, No.12, 8th June 1978, pp366-367

[7] Sarhadi, M.
"Spectral analysis at high frequencies using a modified FFT."
Proc. 1989 International Symposium on Computer Architecture and Digital Signal Processing.
Hong Kong, 11-14 Oct. 1989, Vol. 1, pp242-246

[8] Bilinsky, I. Ya; Vystavkin, A. N. & Mikelson, A. K.
"Processing of randomly-sampled signals." in
'Signal Processing III: Theories and Applications.'
Young, I. T. et al. (editors)
Proc. EUSIPCO-86: Third European Signal Processing Conference.
The Hague, Netherlands, 2-5 Sept. 1986, Vol. 1, pp109-112

[9] Beutler, F. J.
"Alias-free randomly timed sampling of stochastic processes."
IEEE Transactions on Information Theory, Vol. IT-16, No.2, March 1970, pp147-152

[10] Masry, E.
"Alias-free sampling: An alternative conceptualization and its applications."
IEEE Transactions on Information Theory, Vol. IT-24, No.3, May 1978, pp317-324

[11] Marvasti, F. A.
"Spectral analysis of random sampling and error free recovery by an iterative method."
Transactions of the Institute of Electronics and Communication Engineers of Japan. Section E, Vol. E69, No.2, February 1986, pp79-82

[12] Marvasti, F. A.
"Spectrum of nonuniform samples."
Electronics Letters, Vol. 20, No.21, 11th October 1984, pp896

[13] Wiley, R. G.
"Recovery of bandlimited signals from unequally spaced samples."
IEEE Transactions on Communications, Vol. COM-26, No.1, January 1978, pp135-137

[14] Masry, E.
"Random sampling and reconstruction of spectra."
Information and Control, Vol. 19, No.4, 1971, pp275-288

[15] Widrow, B. & Stearns, S. D.
'Adaptive Signal Processing.'
Prentice-Hall, Englewood Cliffs, New Jersey, 1985

[16] Knuth, D. E.
'The Art of Computer Programming.'
Vol. 1, "Fundamental Algorithms," &
Vol. 2, "Seminumerical Algorithms."
Second Edition. Addison-Wesley, Reading, Massachusetts, 1973

[17] Liu, B. (editor)
'Digital Filters and The Fast Fourier Transform.'
Benchmark Papers in Electrical Engineering and Computer Science, Vol 12
Dowden, Hutchinson & Ross Inc., Stroudsburg, Pennsylvania, 1975

[18] Pipes, L. A. & Hovanessian, S. A.
'Matrix-Computer Methods in Engineering.'
Wiley, New York, 1969

[19] Press, W. H; Flannery, B. P; Teukolsky, S. A. & Vettering, W. T.
'Numerical Recipes.'
Cambridge University Press, Cambridge, 1986

[20] Proakis, J. G. & Manolakis, D. G.
'Introduction to Digital Signal Processing.'
Macmillan, New York, 1988