Efficient Narrow-band Notch Filter

by

James W. Thomas

Project and Report submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

APPROVED:

Dr. F. J. Ricci, Chairman

Dr. Joseph Knight

Timothy R. Cooper

April 1994

Fairfax, Virginia

Efficient Narrow-band Notch Filter

by

James W. Thomas

Committee Chairman: Dr. F. J. Ricci

Electrical Engineering

(ABSTRACT)

An efficient digital narrow-band notch filter may be realized by subtracting weighted sine and cosine vectors from the input signal. The weights of the sine and cosine vectors are determined using adaptive filtering techniques while the frequency of these vectors remains fixed. The frequency and phase response of a filter constructed in this manner is absolutely flat within the passband, while the stopband rejection is 95 dB. The computational complexity of this filter is equivalent to that of a four-tap FIR filter.

Efficient Narrow-band Notch Filter

A common problem encountered in digital audio recording systems is the unwanted presence of 60 Hz hum. The traditional approach to eliminating this hum is to employ either an FIR or IIR filter with a notch at 60 Hz. If the bandwidth of the stopband is very narrow, the number of taps required by an FIR filter can easily exceed 100, and a system a requiring linear phase response cannot use an IIR filter. The approach presented in this paper is to use adaptive filtering techniques to remove the 60 Hz component by subtracting weighted sine and cosine vectors from the input signal. The weights of the sine and cosine vectors are adaptively adjusted to minimize the energy in the output signal. The frequency of the sine and cosine vectors is fixed at 60 Hz. A block diagram illustrates this process in Figure 1. This approach should produce a very narrow band filter with linear phase response.

I. Derivation

Adaptive filters are usually constructed by minimizing the difference between a predicted signal and a received signal. This difference is called the error signal. The predictor is often an FIR or IIR filter with a known input

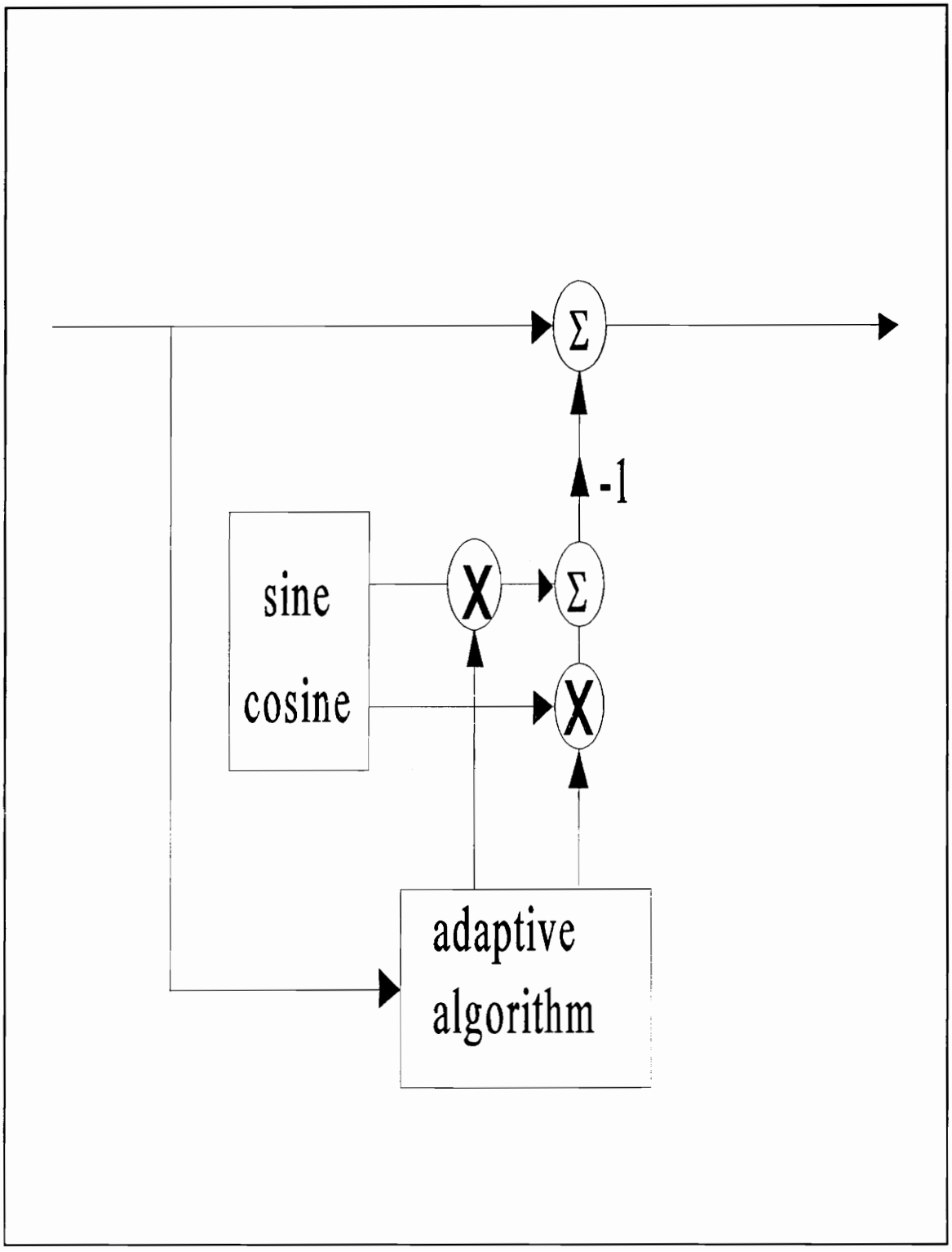

Figure 1: Block Diagram of Notch Filter

as its stimulus. For a typical FIR system, the following system equation is employed:

$$e=y-(Xh)$$

where e is the error vector, y is the received vector, X is a matrix representation of the input signal, and h is the coefficient vector. The energy in the error vector, (e,e), can then be minimized by setting its derivative with respect to h equal to 0 and solving for h. This yields

$$h=(X^H X)^{-1} X^H y .$$

There are several methods of solving this equation, but this has been well treated in other texts[1] and will not be presented here. The approach described in this paper differs significantly in that it is not implemented by convolution. Therefore, the equations for this filter must be derived. The system equation is

$$y=x-\alpha_1 c-\alpha_2 s$$

where α_1 and α_2 are the weights of c, the cosine vector, and s, the sine vector. The energy in the output vector is thus:

$$(y,y)=(x-\alpha_1 c-\alpha_2 s, x-\alpha_1 c-\alpha_2 s)$$

$$(y,y)=(x,x)-\alpha_1(x,c)-\alpha_2(x,s)-\alpha_1(c,x)+\alpha_1\alpha_2(c,s)+\alpha_1^2(c,c)-\alpha_2(s,x)+\alpha_1\alpha_2(s,c)+\alpha_2^2(s,s)$$

however, c and s are orthogonal, and if x is assumed to be a real sequence,

$$(y,y) = (x,x) - 2\alpha_1 (x,c) - 2\alpha_2 (x,s) + \alpha_1^2 (c,c) + \alpha_2^2 (s,s)$$

Taking the derivative of (y,y) with respect to α_1 yields,

$$\frac{d(y,y)}{d(\alpha_1)} = 2\alpha_1(x,c) - 2(x,c)$$

Setting this result to zero and solving for α_1 yields

$$\alpha_1 = \frac{(x,c)}{(c,c)}$$

Similarly,

$$\alpha_2 = \frac{(x,s)}{(s,s)}$$

It should come as no surprise that these results are identical to the real and imaginary outputs of the discrete Fourier transform (DFT) scaled to adjust for the length of the sine and cosine vectors.

II. Implementation

Calculating the DFT by evaluating the dot product of the input sequence with sine and cosine functions is not the most efficient method of arriving at the values of α_1 and α_2. The Goertzel algorithm has been shown[2] to require fewer coefficients and fewer multiplies to achieve the same result. Even so, having fewer multiplies is not necessarily an advantage using the DSP chips available today, because most of these devices are capable of performing a simultaneous multiplication and addition. However, requiring only two coefficients (one real and one complex)

is enough of an advantage to warrant the use of the Goertzel algorithm even by a DSP chip. Also, since the software written for this project was developed on a personal computer lacking a DSP chip, the Goertzel algorithm provided a significant speed improvement.

A block diagram of the Goertzel algorithm is shown in Figure 2. Because a Goertzel filter calculates a single bin of an N-point DFT, the frequency resolution of each bin is $bw=\frac{f_s}{N}$. To determine the value of N for a particular frequency resolution, this equation must be solved for N: $N=\frac{f_s}{bw}$. Thus, for a resolution of one Hertz, the length of the Goertzel filter must be equal to the sampling frequency. At first glance, it might seem that using a Goertzel filter of this length would cause the number of computations in the overall filter to skyrocket. However, by implementing the overall filter in a block mode rather than in a sample mode, this problem is mitigated. In other words, the Goertzel calculation is performed on a block of input samples, and the result is used to subtract out the interference signal for the entire block of data. This assumes that the interference signal does not change very much from one block to the next. Also, implementing this filter in block mode will incur a delay equal to 1/bw. For a one Hertz filter,

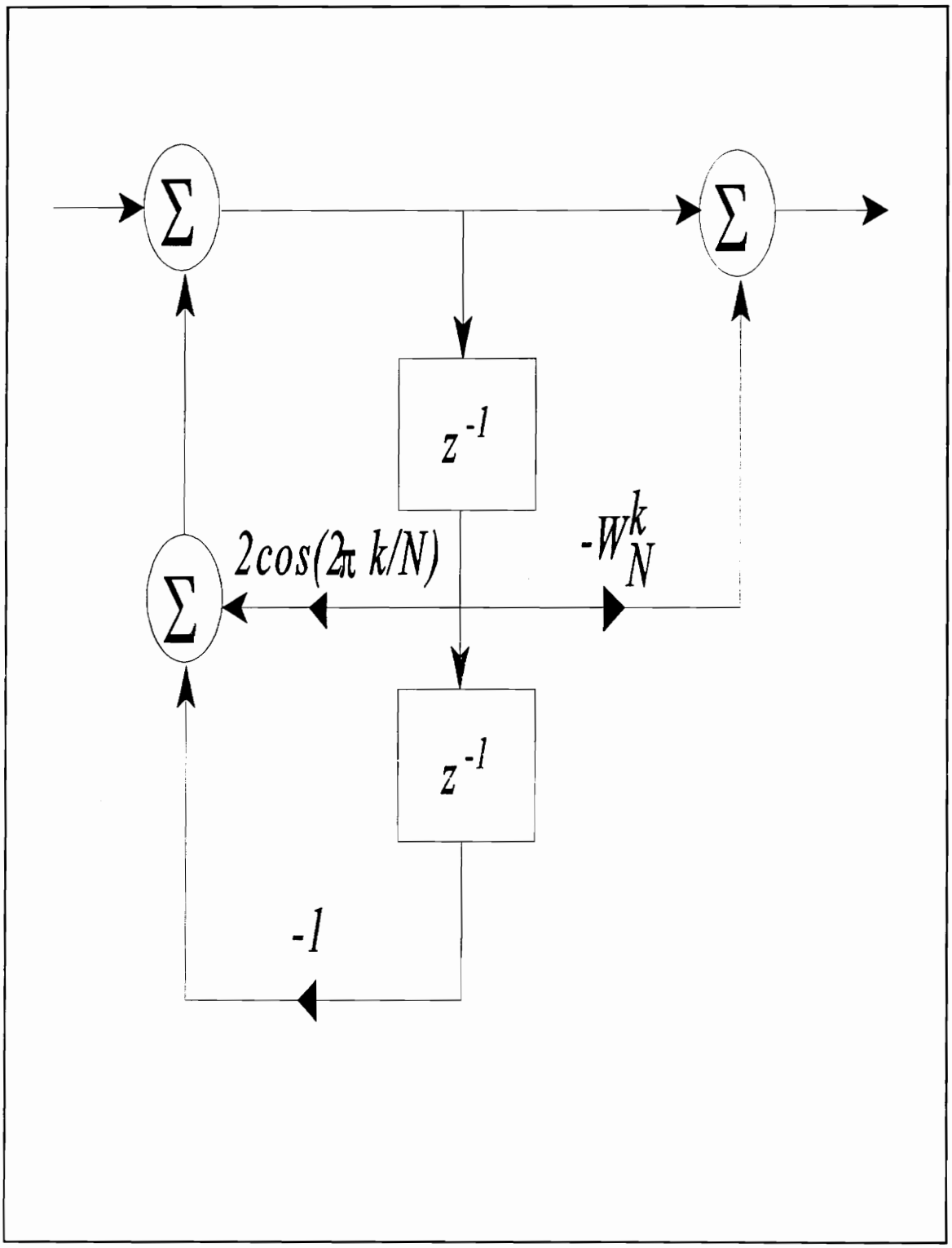

Figure 2: Block Diagram of the Goertzel Algorithm

this delay is one second.

Following thedefinition of the algorithm, the software to implement it was written. I chose to write the software in ANSI C because all the major DSP chips have C compilers. The software is shown in listings 1 through 6.

Listings 1 and 2 show the implementation of the Goertzel algorithm. The first listing is of the header file which contains the function definitions and data types. The second contains the actual C code. Two functions were written in the goertzel module, open_goertzel() and goert(). The open_goertzel() function allocates and initializes a goertzel object. The goertzel object contains all the information which must be retained by goert() from one call to the next. The goert() function is the workhorse portion of the module. It evaluates the input data as per the Goertzel algorithm and updates the goertzel object.

Listings 3 and 4 the contain the header file and C code for the notch filter itself. The C code consists of three functions, open_notch(), notch_filter(), and main(). The open_notch() function creates and initializes all the resources required by notch_filter(). Since notch_filter() requires the services of goert(), open_notch() calls

open_goertzel(). The notch_filter() function calls the goert() function, scales the output of the DFT and subtracts the weighted sine and cosine vectors from the input signal. It should be noted here that the efficiency of the filter would greatly increase if the sine and cosine evaluations were replaced with lookup tables. This was not done in notch_filter() due to the memory limitations of the machine on which it was developed. The main() function evaluates the input arguments to the program, opens the input and output files, calls open_notch(), and enters a loop which processes all the input samples. In the loop, it reads the input samples, calls the notch_filter() function, and writes the output samples to the output file.

Listing 5 contains the routines used for reading samples from the input file and writing samples to the output file. The file format of the sample data is compatible with the PC-DSP program[3] which was used to evaluate the filter. Each sample is represented in ASCII as a floating point number. Samples are delimited by a carriage-return/line-feed. As there is nothing extraordinary about these routines, they will not be discussed here.

III. Performance

A digital filter can be completely characterized by its impulse response. Its frequency response is simply the magnitude of the Fourier transform of its impulse response. Its phase response is also derived from the Fourier transform. The frequency and phase responses of the filter presented in this paper are shown in Figures 3 and 4. These were computed by injecting an impulse into the filter and performing an FFT on the filter's output. The frequency and phase responses were then derived from the output of the FFT. The magnitude of the 60 Hz component is 1.75×10^{-5} which provides a 95 dB rejection of the interference signal. Both the 59 Hz and 61 Hz components are at 0 dB. The phase is flat within the passband.

If the interference frequency is not synchronous to the sample clock, a frequency delta will be introduced. To test the effect of this, a 60.1 Hz signal was injected into the filter. It was rejected by 27 dB. This is a significant jump in the stopband rejection. To get an overall picture of this effect, the frequency and phase responses were computed at a higher resolution. In this case, an impulse of length 8192 was injected into the system which was set to sample at 512 Hertz. Then an 8192-point FFT was computed

and from this, the frequency and phase responses were calculated. These results are shown in Figures 5 and 6. It can be seen that the stopband of this filter is extremely narrow, and it would not take much of a delta for the interference signal to drift out of the stopband. Although not implemented here, a possible solution to this problem would be to determine the frequency drift by comparing the results of successive Goertzel computations. If the interference frequency is not exactly 60 Hz, these results will vary by a constant amount from one block to the next. If the amplitude of the interference signal is not changing, the sum of the squares of α_1 and α_2 will remain constant. Once the phase drift between successive blocks has been determined, the frequency drift could be calculated and fed back into both the Goertzel algorithm and the subtraction portion of the filter.

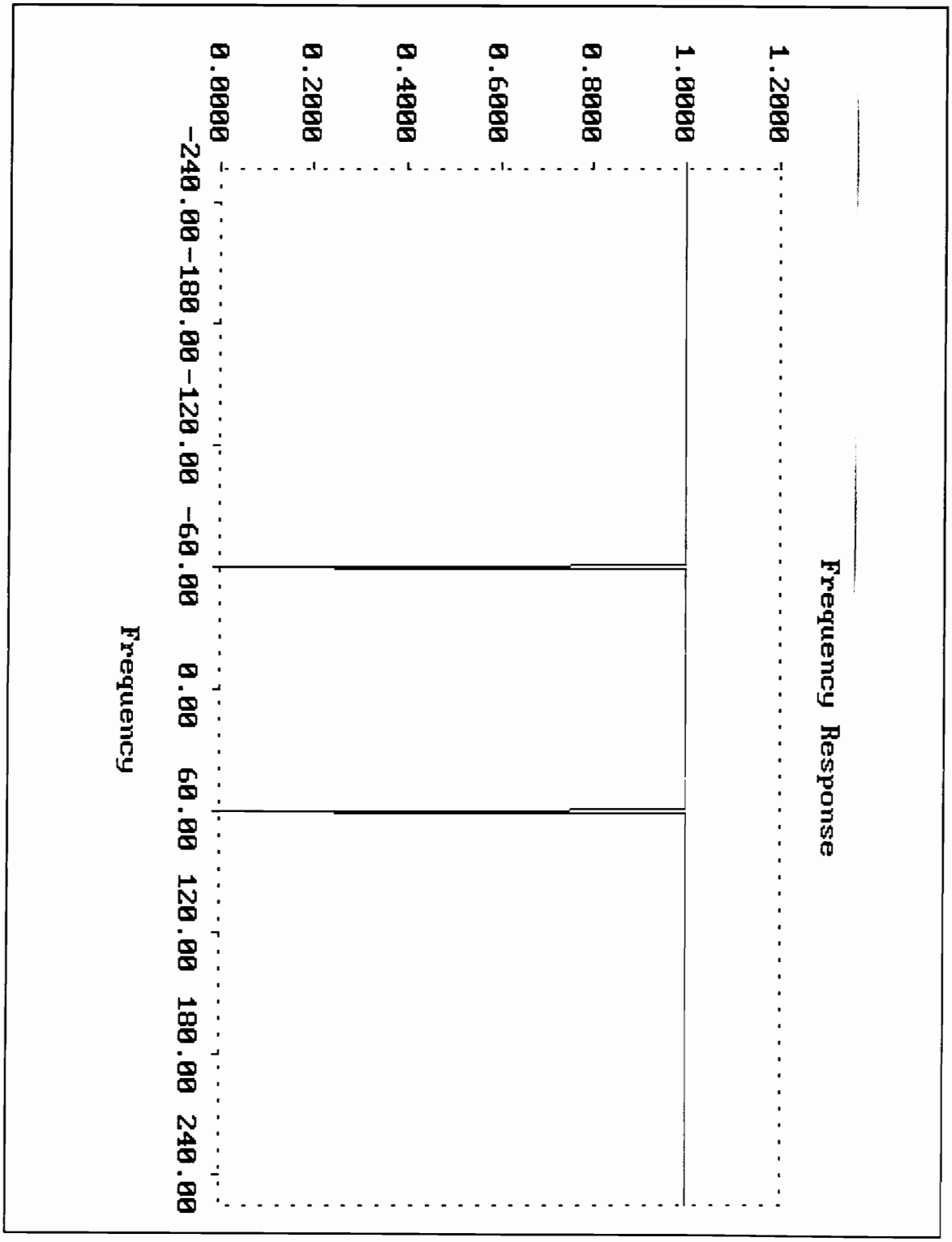

Figure 3: Frequency Response of the Notch Filter (Linear Scale).

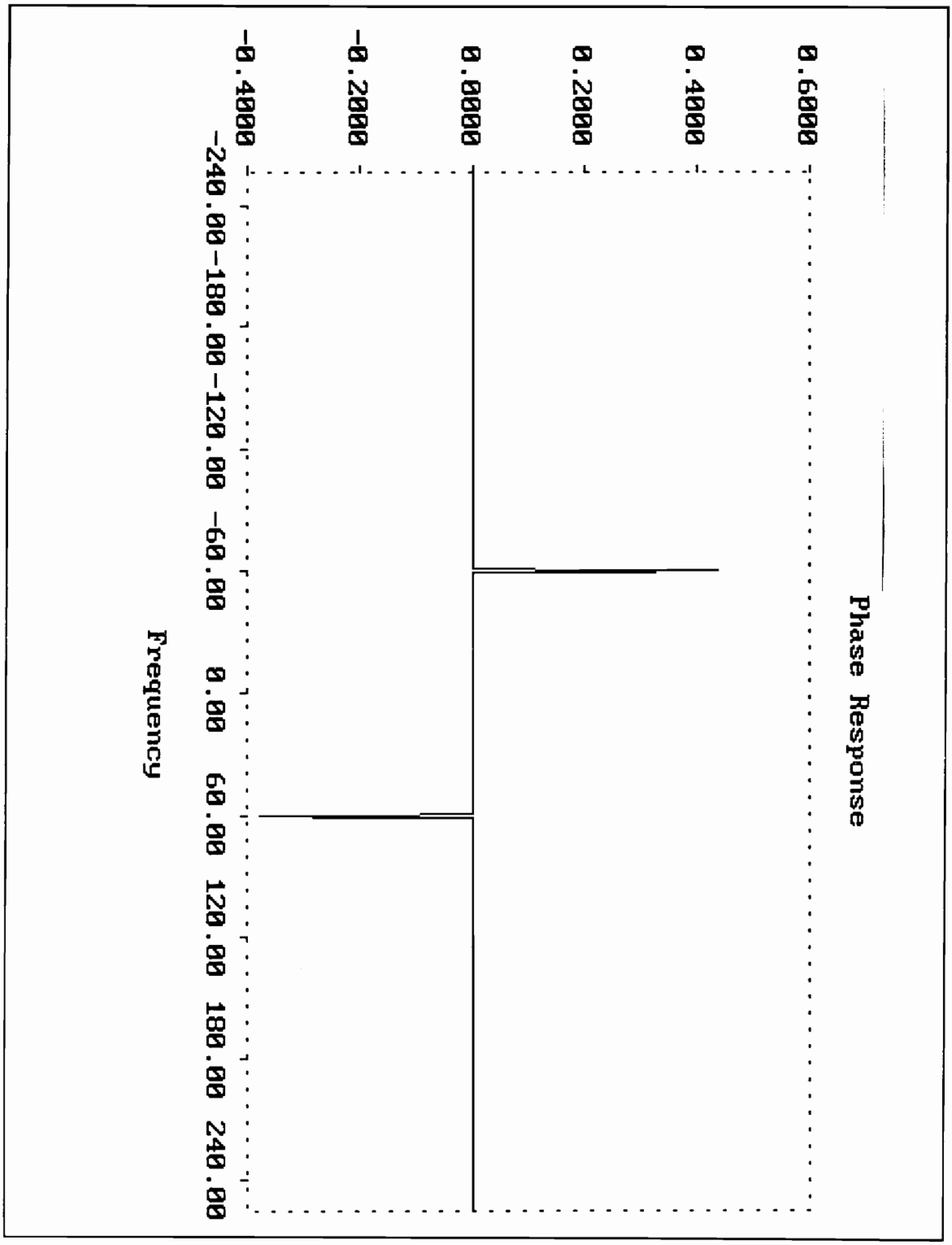

Figure 4: Phase Response of the Notch Filter

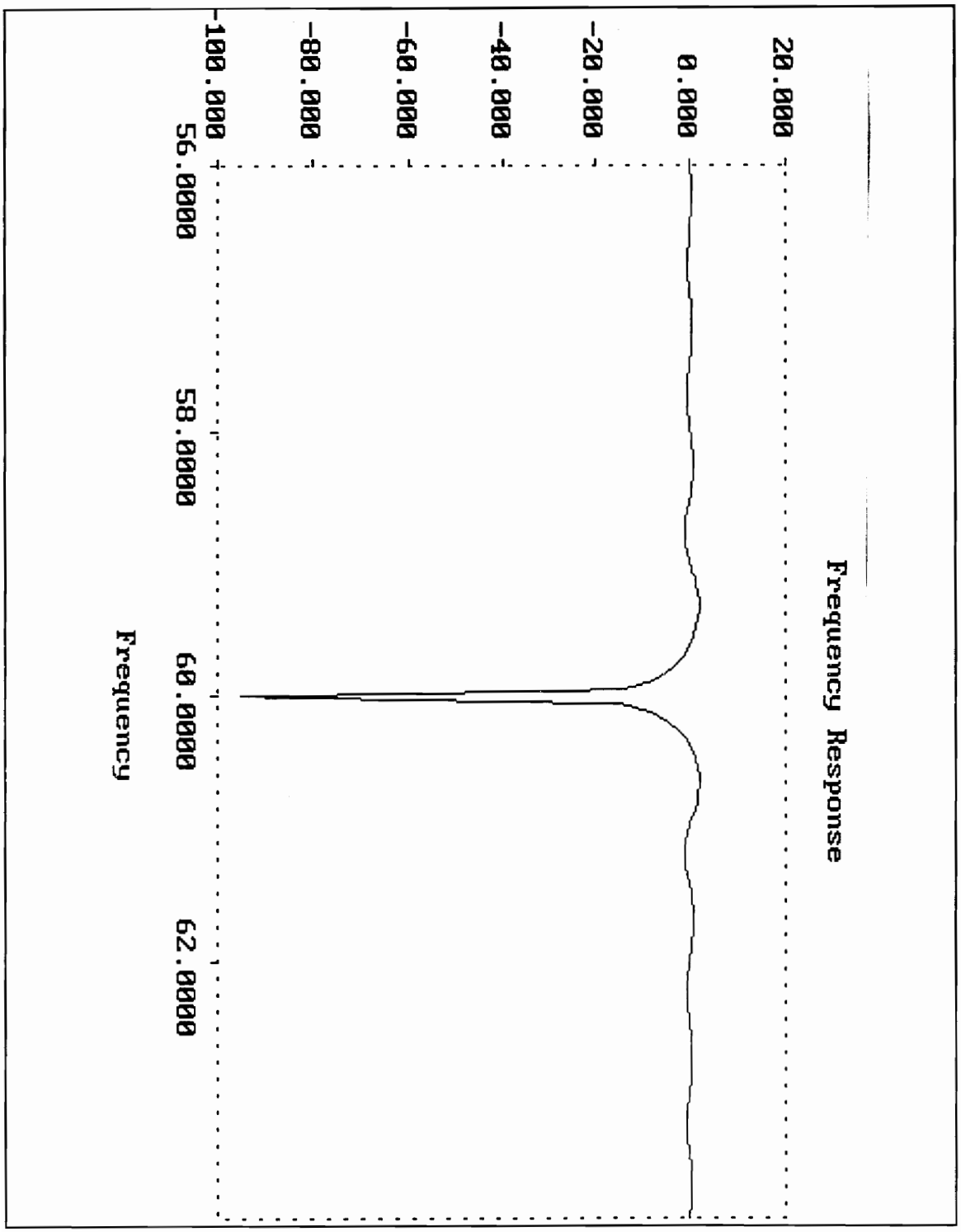

Figure 5: Expanded View of the Frequency Response (Log Magnitude)

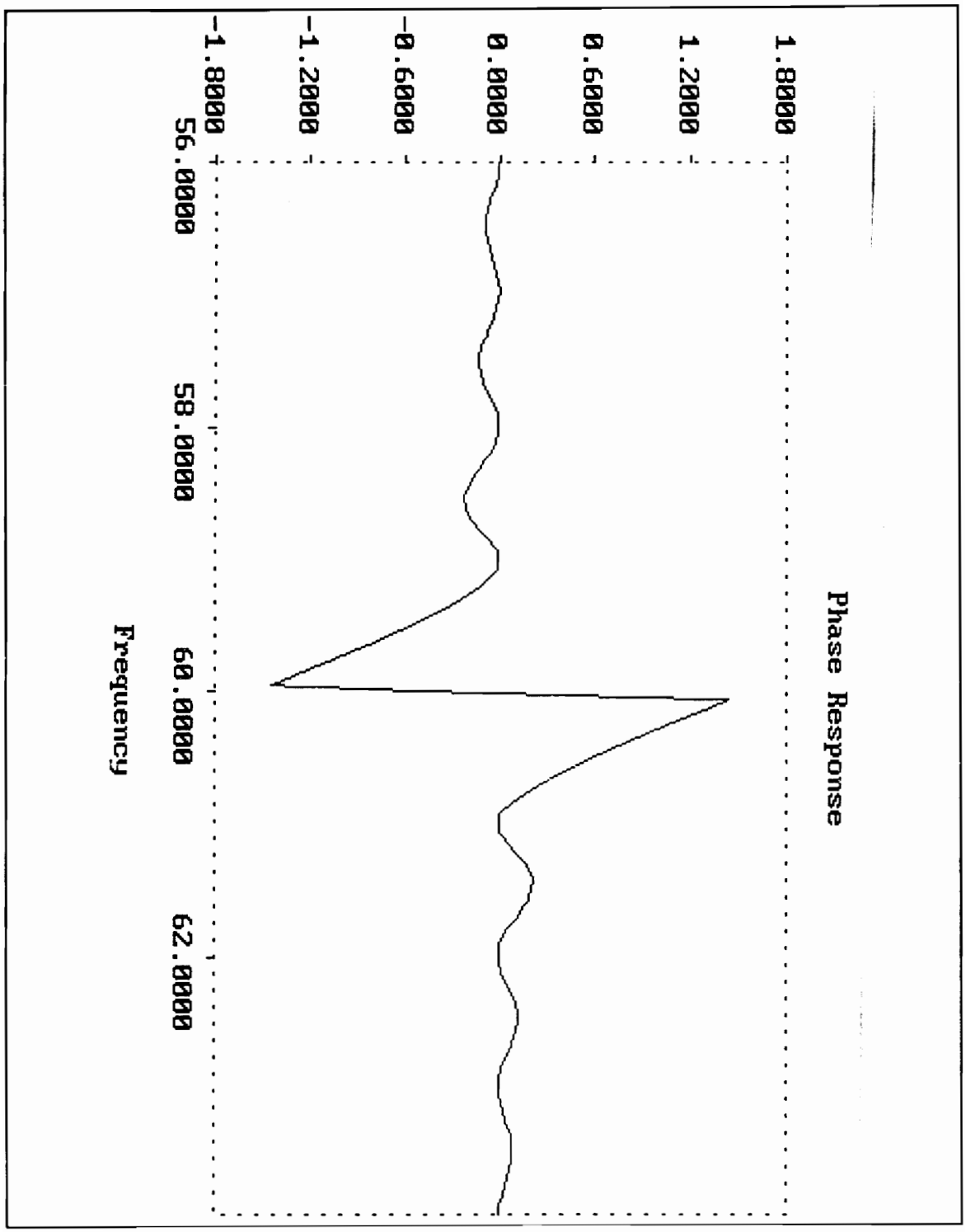

Figure 6: Expanded View of Phase Response.

IV. Conclusions

This filter's computational efficiency compared to an FIR filter is far superior. The number of instruction cycles per sample for the filter can be expressed as $I=\frac{2f_s+k}{f_s}+2$, where f_s is the sample rate and k is the number of instruction cycles in the Goertzel feed forward phase. This assumes that the both the Goertzel loop and the subtraction loop can be implemented in two instruction cycles. This is typical of today's powerful DSP chips. If k is small compared to f_s, $I \approx 4$. Most DSP chips are capable of calculating a single FIR filter tap in one instruction, so for an FIR filter, $I \approx N$, where N is the number of taps in the FIR filter. Therefore, an FIR filter with equivalent computational efficiency can have only four taps. I think it is safe to say that it is not possible to achieve 95 dB rejection of a one Hertz stopband with 400 taps, much less with only four.

A two pole IIR filter could be constructed to notch out the 60 Hz interference signal by placing complex conjugate zeroes on the unit circle at the interference frequency and by placing complex conjugate poles just inside the unit circle (at the same angle). Such a filter would consist of

four multiplications and four additions per sample. This could be implemented in a four cycle loop on a modern DSP chip. A filter constructed in this way would require a comparable number of instruction cycles to implement, but would not provide linear phase response and would also introduce passband ripple problems.

An application requiring a narrowband notch filter with linear phase and limited processing resources would likely benefit from the approach presented in this paper.

Listing 1.

```
/*************************************************************/
/* Filename: goert.h        */

#ifndef _GOERT_
#define _GOERT_
typedef struct
{
   float real;
   float imag;
} Complex;

typedef struct
{
   float    a;
   Complex  b;
   int      n;
   int      into;
   float    v1;
   float    v2;
} Goertzel;

Goertzel *open_goertzel(int freq, int n, int fs);
int goertzel(float *x, int size, Complex *y, Goertzel
*goert);
void close_goertzel(Goertzel *attrs);

#endif
```

Listing 2.

```
/*************************************************************/
/* Filename: goert.c */


#include <math.h>
#include <stdlib.h>
#include "goert.h"
#define TWO_PI    (2.0*3.1415926536)

void close_goertzel(attrs)
Goertzel *attrs;
{
   free((void*)attrs);
}

Goertzel *open_goertzel(freq, n, fs)
int   freq;
int   n;
int   fs;
{
   Goertzel *goert;
   float    angle;
   float    k;

   if((goert = (Goertzel*)malloc(sizeof(Goertzel))) ==
(Goertzel*)0)
   {
      printf("Could not allocate memory for a Goertzel\n");
      exit(-1);
   }
   k = (float)freq/(float)fs;
   angle = TWO_PI * k;
   goert->b.real = cos(angle);
   goert->b.imag = sin(angle);
   goert->a = 2.0*goert->b.real;
   goert->n = n;
   goert->into = 0;
   goert->v1 = 0.0;
   goert->v2 = 0.0;
   return(goert);
}

int goertzel(x, size, y, goert)
float    *x;
int      size;
```

```
Complex  *y;
Goertzel *goert;
{
   int   i;
   int   j,k;
   int   iters;
   int   done;
   float v0, v1, v2;


   iters = goert->n - goert->into;
   iters = (iters > size) ? size : iters;
   goert->into += iters;
   v1 = goert->v1;
   v2 = goert->v2;

   for(i=0; i < iters; i++)
   {
      v0 = *x++ + goert->a * v1 - v2;
      v2 = v1;
      v1 = v0;
   }
   if (goert->into == goert->n)
   {
      y->real = goert->b.real * v1 - v2;
      y->imag = -goert->b.imag * v1;
      goert->into = 0;
      goert->v1 = 0.0;
      goert->v2 = 0.0;
      done = 1;
   }
   else
   {
      goert->v1 = v1;
      goert->v2 = v2;
      done = 0;
   }
   return(done);
}
```

Listing 3.

```
/*************************************************************/
/* Filename: nf.h */

#ifndef _NF_
```

```
#define _NF_

#ifndef NULL
#define NULL (0)
#endif

#ifndef TWO_PI
#define TWO_PI (2.0*M_PI)
#endif

#include "goert.h"
typedef struct
{
   Goertzel *goert;
   int      size;
   float    scale;
   float    angle;
   float    phase;
   Complex  last;
} Notch_Attrs;

Notch_Attrs *open_notch(int freq, int len, int fs);
float notch_filter(float *x, float *y, int size, Notch_Attrs
*attrs);
#endif
```

Listing 4.

```
/*************************************************************/
/* Filename: nf.c */


#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "samples.h"
#include "nf.h"
#include "goert.h"
#define  FS   512

#define ARGC 5

#define IN_ARG 1
#define OUT_ARG 2
#define NOTCH_ARG 3
#define FS_ARG 4

Notch_Attrs *open_notch(freq, len, fs)
int   freq;
int   len;
int   fs;
{
   Notch_Attrs *attrs;

   if ((attrs = (Notch_Attrs*)malloc(sizeof(Notch_Attrs)))
       == (Notch_Attrs*)NULL)
   {
      return(attrs);
   }
   else if ((attrs->goert = open_goertzel(freq, len, fs)) ==
(Goertzel*)NULL)
   {
      free((void*)attrs);
      return((Notch_Attrs*)NULL);
   }
   else
   {
      attrs->size = len;
      attrs->scale = (fs==0) ? 1.0 : 2.0 / (float)fs;
      attrs->angle = M_PI * (float)freq * attrs->scale;
      attrs->phase = 0.0;
      attrs->last.real = 0.0;
```

```
      attrs->last.imag = 0.0;
      return(attrs);
   }
}


float notch_filter(x, y, size, attrs)
float       *x;
float       *y;
int         size;
Notch_Attrs *attrs;
{
   Complex comp;
   float   angle;
   float   mag;
   float   *x_ptr;
   int     i;
   int     done;
   float   real, imag;

   if (done = goertzel(x, size, &comp, attrs->goert))
   {
      attrs->last.real = attrs->scale * comp.real;
      attrs->last.imag = attrs->scale * comp.imag;
   }
   x_ptr = x;
   for(i = 0; i < size; i++)
   {
      real = attrs->last.real * cos(attrs->phase);
      imag = attrs->last.imag * sin(attrs->phase);
      *y++ = *x_ptr++ - real - imag;
      attrs->phase += attrs->angle;
      if (attrs->phase > 2.0*M_PI) attrs->phase -= 2.0*M_PI;
   }
   if (done) attrs->phase = 0.0;
   return (done);
}


/* argv[0] = executable name */
/* argv[1] = input file      */
/* argv[2] = output file     */
/* argv[3] = notch freq      */
/* argv[4] = sample rate     */

static char default_in[] = {"infile.dat"};
```

```
static char default_out[] = {"outfile.dat"};
main(argc, argv)
int  argc;
char **argv;
{
   float       *x;
   float       *y;
   int         size, fs, notch, samples_read, i;
   Notch_Attrs *n_attrs;
   char        *in_name, *out_name;
   FILE        *infile, *outfile;



   if (argc != ARGC)
   {
      in_name = default_in;
      out_name = default_out;
      fs = FS;
      printf("Using default parameters:\n");
   }
   else
   {
      in_name = argv[IN_ARG];
      out_name = argv[OUT_ARG];
      fs = atoi(argv[FS_ARG]);
      notch = atoi(argv[NOTCH_ARG]);
      printf("Using supplied parameters\n");
   }
   size = fs;
   printf("input file: %s, output file: %s\n",in_name,
out_name);
   printf("sample rate %d\n", fs);

   if ((outfile = fopen(out_name, "w")) == NULL)
   {
      printf("Could not open %s for output\n", out_name);
      exit(-1);
   }
   else if ((infile = fopen(in_name, "r")) == NULL)
   {
      printf("Could not open %s for input\n", in_name);
      exit(-1);
   }
   else if ((x = (float*)malloc(sizeof(float)*size)) ==
(float*)NULL)
```

```
   {
      printf("Not enough memory to create input buffer\n");
      exit(-1);
   }
   else if ((y = (float*)malloc(sizeof(float)*size)) ==
(float*)NULL)
   {
      printf("Not enough memory to create output buffer\n");
      exit(-1);
   }
   else
   {
      n_attrs = open_notch(notch, fs, fs);

      i = 0;
      while (printf("reading %d...  \r", i),
             samples_read = get_samples(infile, x, size))
      {
         printf("filtering %d... \r", i);
         notch_filter(x, y, samples_read, n_attrs);
         printf("writing %d...   \r", i++);
         put_samples(outfile, y, samples_read);
      }
      fclose(infile);
      fclose(outfile);
      printf("Finished    \n");
   }
}
```

Listing 5.

```
/*************************************************************
**********************/
/* Filename: samples.h */

#ifndef _SAMPLES_
#define _SAMPLES_
/* samples.h */

int get_samples(FILE *infile, float *buf, int size);
void put_samples(FILE *outfile, float *buf, int size);
#endif
```

Listing 6.

```
/*************************************************************
**********************/
/* Filename: samples.c */

#include <stdio.h>
#include <stdlib.h>
#include "samples.h"

int get_samples(infile, buf, size)
   FILE *infile;
   float *buf;
   int size;
{
   int   i;
   char  a[80];

   for (i=0;i<size;i++) {
      if (fgets(a,80,infile) == NULL) break;
      *buf++ = atof(a);
   }
   return (i);
}

void put_samples(outfile, buf, size)
   FILE *outfile;
   float *buf;
   int size;
{
   for (;size > 0;size--) {
      fprintf(outfile,"%f\n",*buf++);
   }
}
```

1.Proakis, John G.; Rader, Charles M.; Ling, Fuyun; and Nikias, Chrysostomos L. *Advanced Digital Signal Processing*, 1992, pp 281-308, MacMillan Publishing Company, New York, NY

2. Proakis, John G., and Manolakis Demitris G. *Introduction to Digital Signal Processing*, 1988, pp 724-726, MacMillan Publishing Company, New York

3. Alkin, Oktay, *PC-DSP*, 1990, Prentice Hall, Englewood Cliffs, New Jersey