Digital Signal Processing 101: Sound Programming for your Workstation

Stephen A. Uhler

Bellcore

sau@bellcore.com

Digital Signal Processing 101: Abstract

With the advent of powerful workstations featuring built-in audio capabilities, many digital signal processing functions that traditionally required dedicated hardware can be easily implemented in software, and still provide real time performance. If you have ever wanted your computer to play songs, help to tune your guitar, or add a little bass boost and reverb to your favorite monologue, you will want to hear about some of the algorithms I will describe. These algorithms will help you create both simple and sophisticated sound applications for your workstation.

Digital Signal Processing Definitions:

- **An interesting or useful analog signal**
- **A digital representation of an analog signal as a sequence of numbers**
- **Numerical computations on a sequence of numbers**

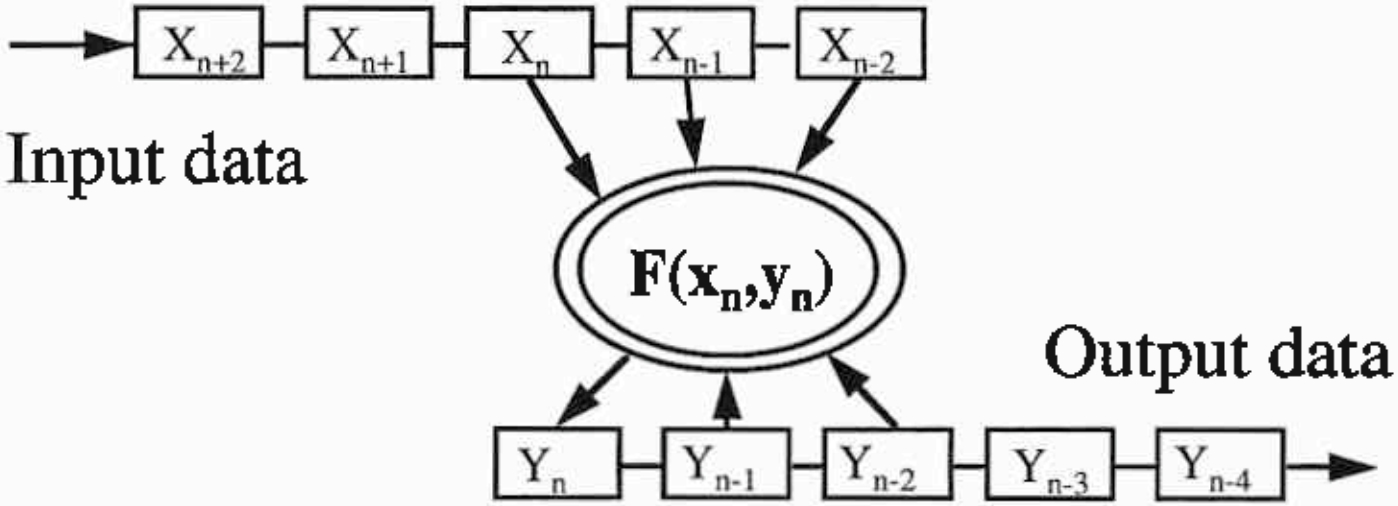
Typical Properties of Digital Signals

- **The data samples fly by quickly**
  - **8000 bytes/sec** μ-Law audio data
  - **160,000 bytes/sec** Hi fidelity stereo
  - **4,500,000 bytes/sec** TV
- **The number of calculations possible per sample is limited**

Definition: A Digital Filter

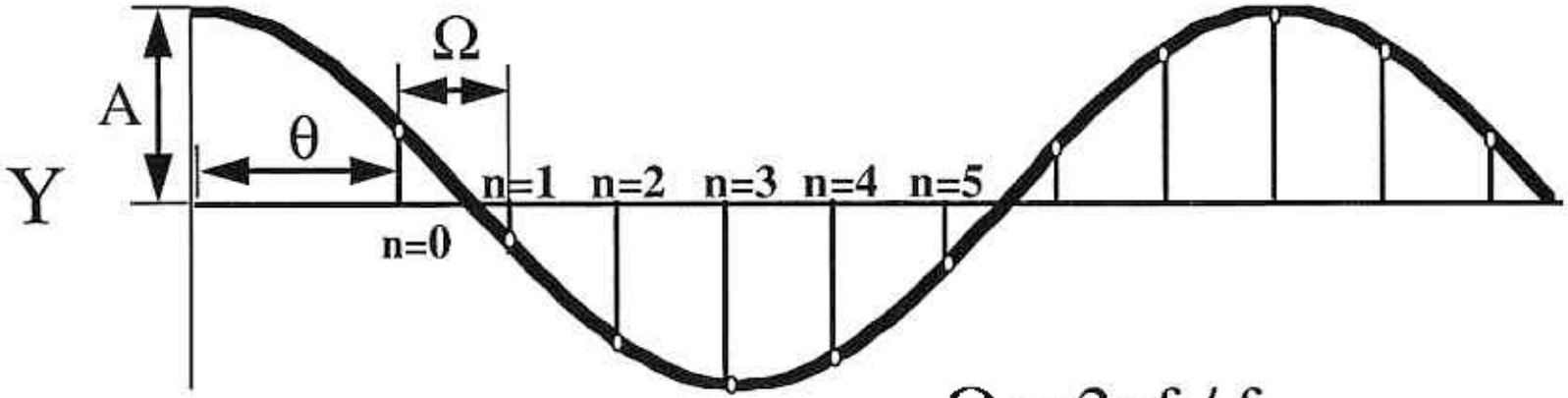
- **A computation on a sequence of input values that produces a sequence of output values**

Recursive Implementation Strategy for a Filter



$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} - (b_1 y_{n-1} + b_2 y_{n-2})$$

Definitions:
Digital Representation of Audio Signals



$$Y_n = A \cos(n\Omega+\theta)$$

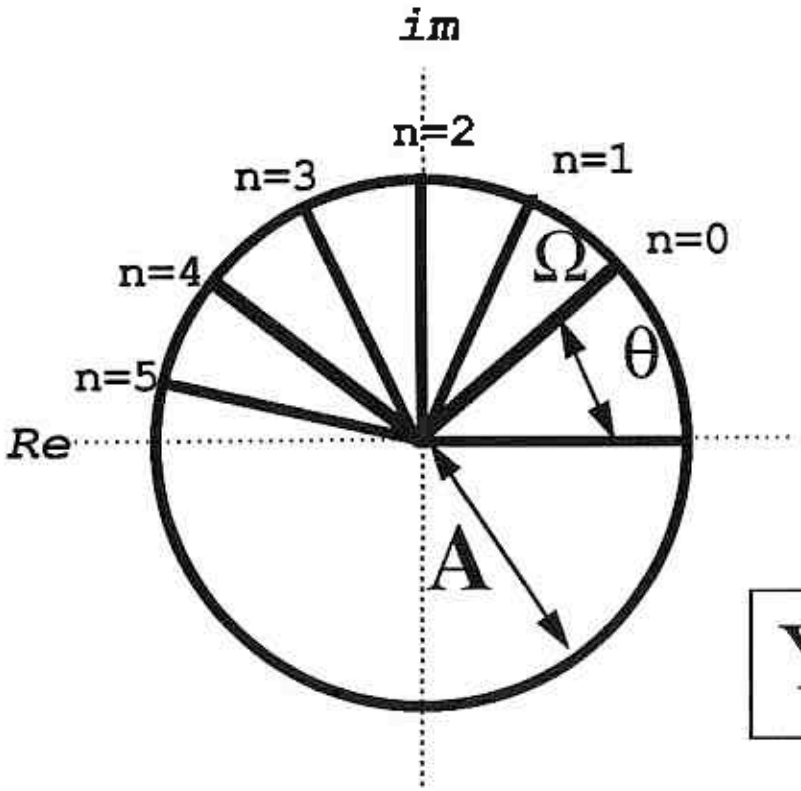
$\Omega = 2\pi f / f_s$

f frequency

f_s sampling frequency

θ initial phase

Representation of Digital Signals: an Alternate View



$\Omega = 2\pi f / f_s$

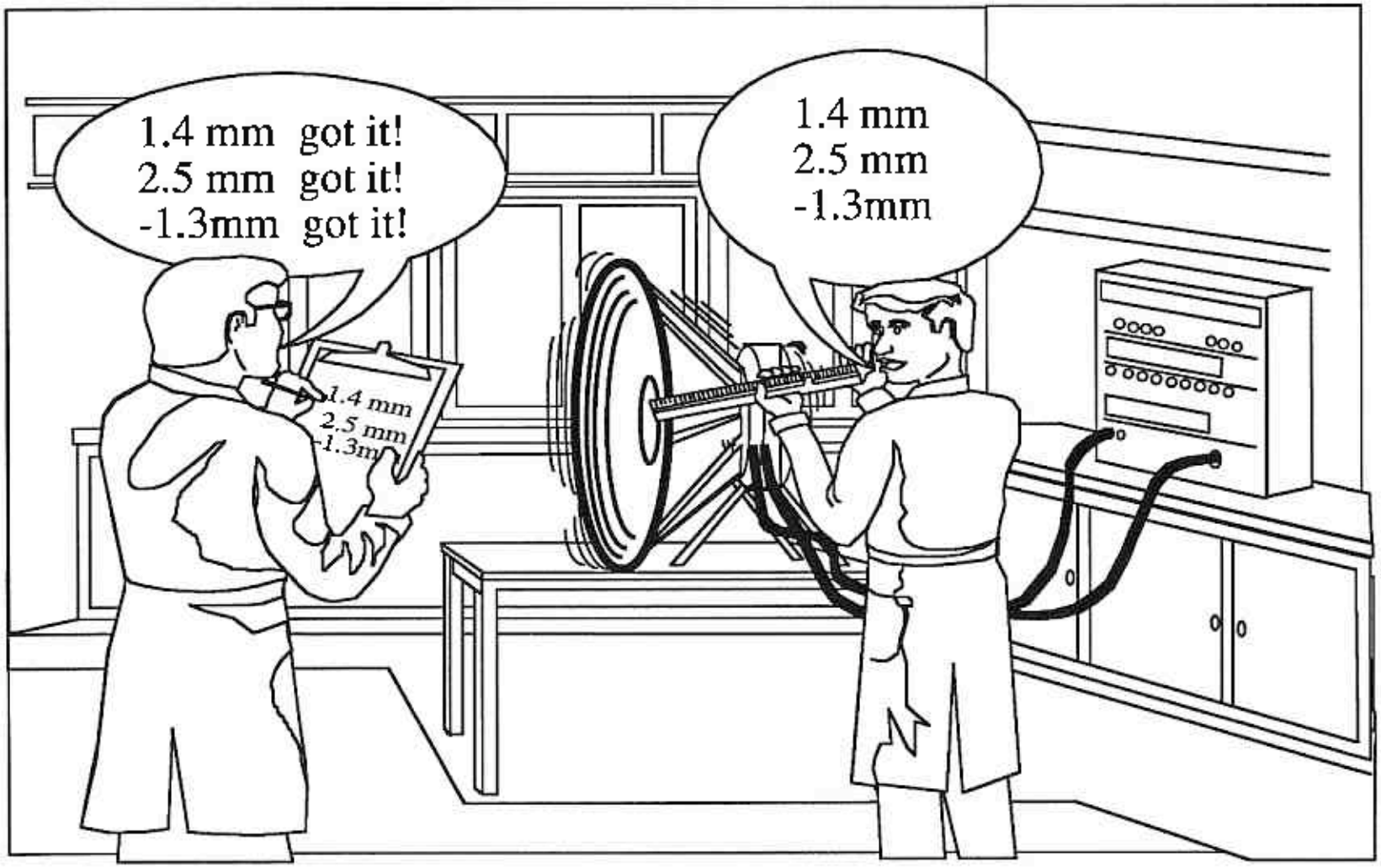
f frequency

f_s sampling frequency

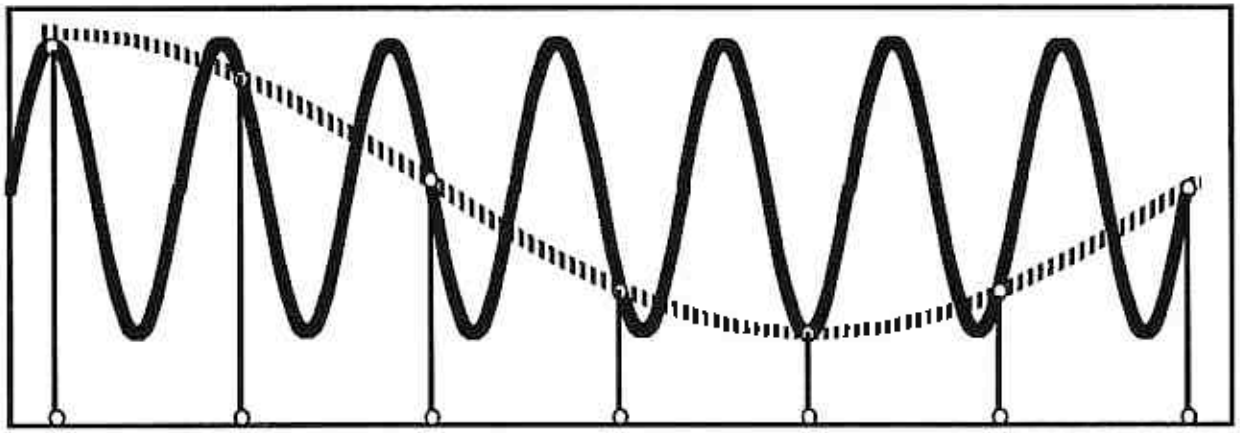
θ initial phase

$$Y_n = A \cos(n\Omega+\theta)$$

Analog to Digital Conversion of an Audio Signal:
Measuring the Displacement of a Speaker Cone



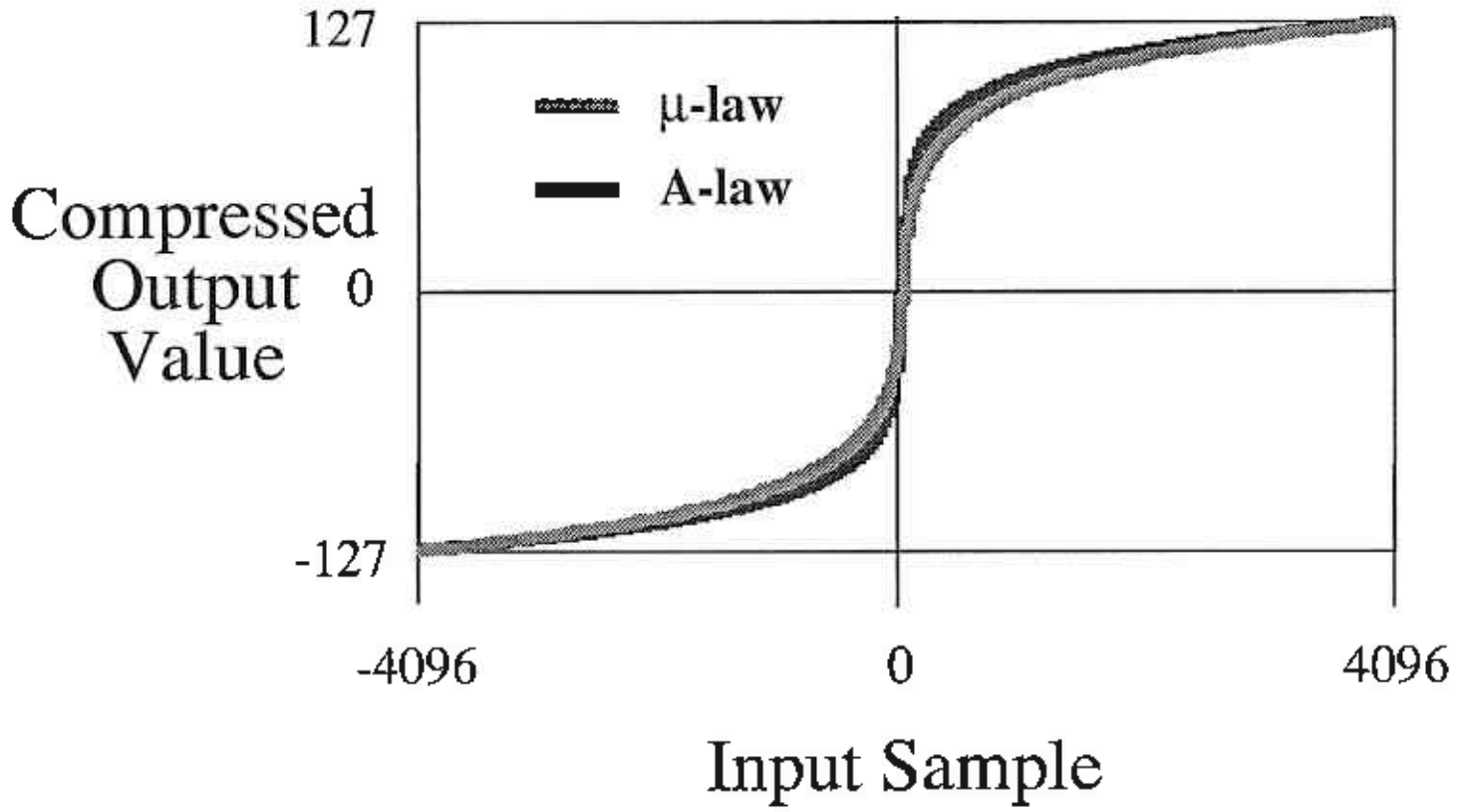
Aliasing Errors in Digital Sampling



Actual Signal

Sampled Signal

Audio Compression:



μ-Law Compression Formula

$$F(x) = \frac{\log(1 + \mu x)}{\log(1 + \mu)}$$

$$\mu = 255$$

```
x     Input value (0 <= x <= 1)
F(x)  Compressed output value
```

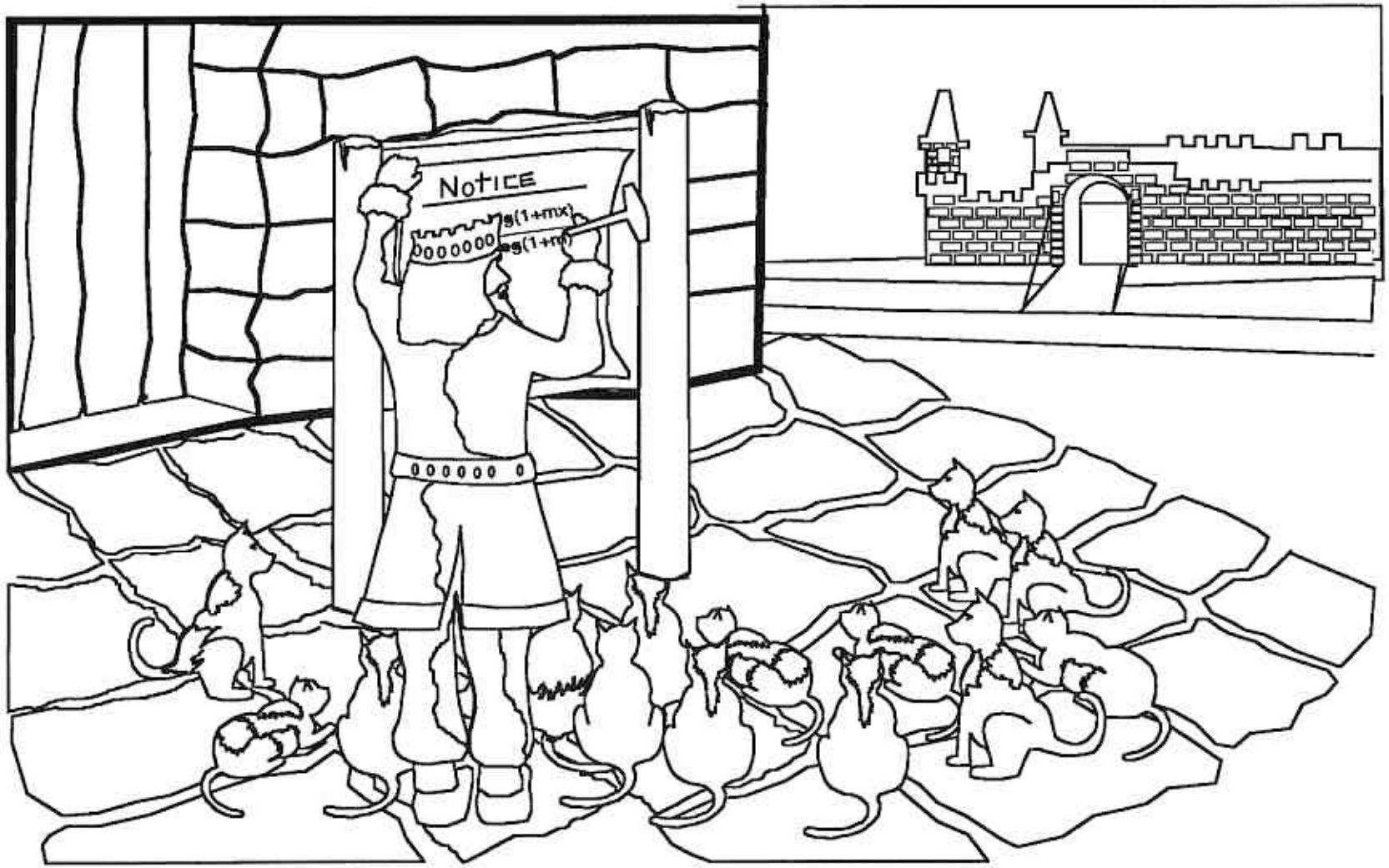
A-Law Compression Formula

$$F(x) = \frac{1 + \log(Ax)}{1 + \log(A)} \qquad x > \frac{1}{A}$$

$$F(x) = \frac{x}{1 + \log(A)} \qquad x < \frac{1}{A}$$

$A = 87.6$

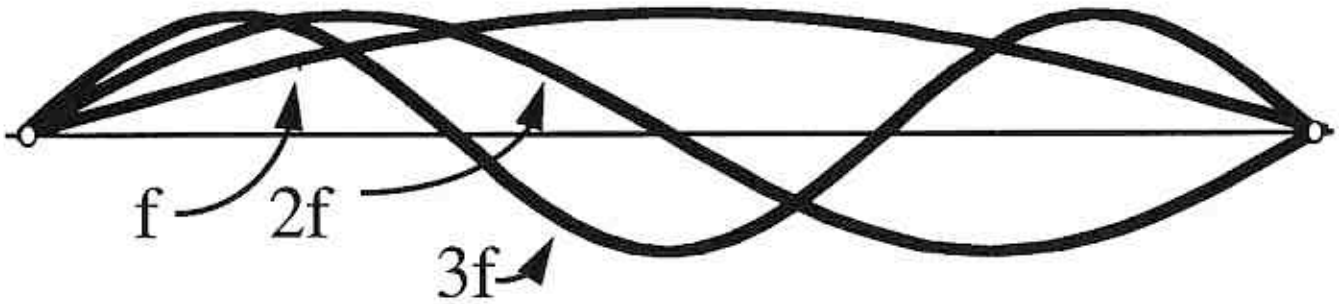
The Discovery of ~~mew~~ μ-Law



sau 6/21/93

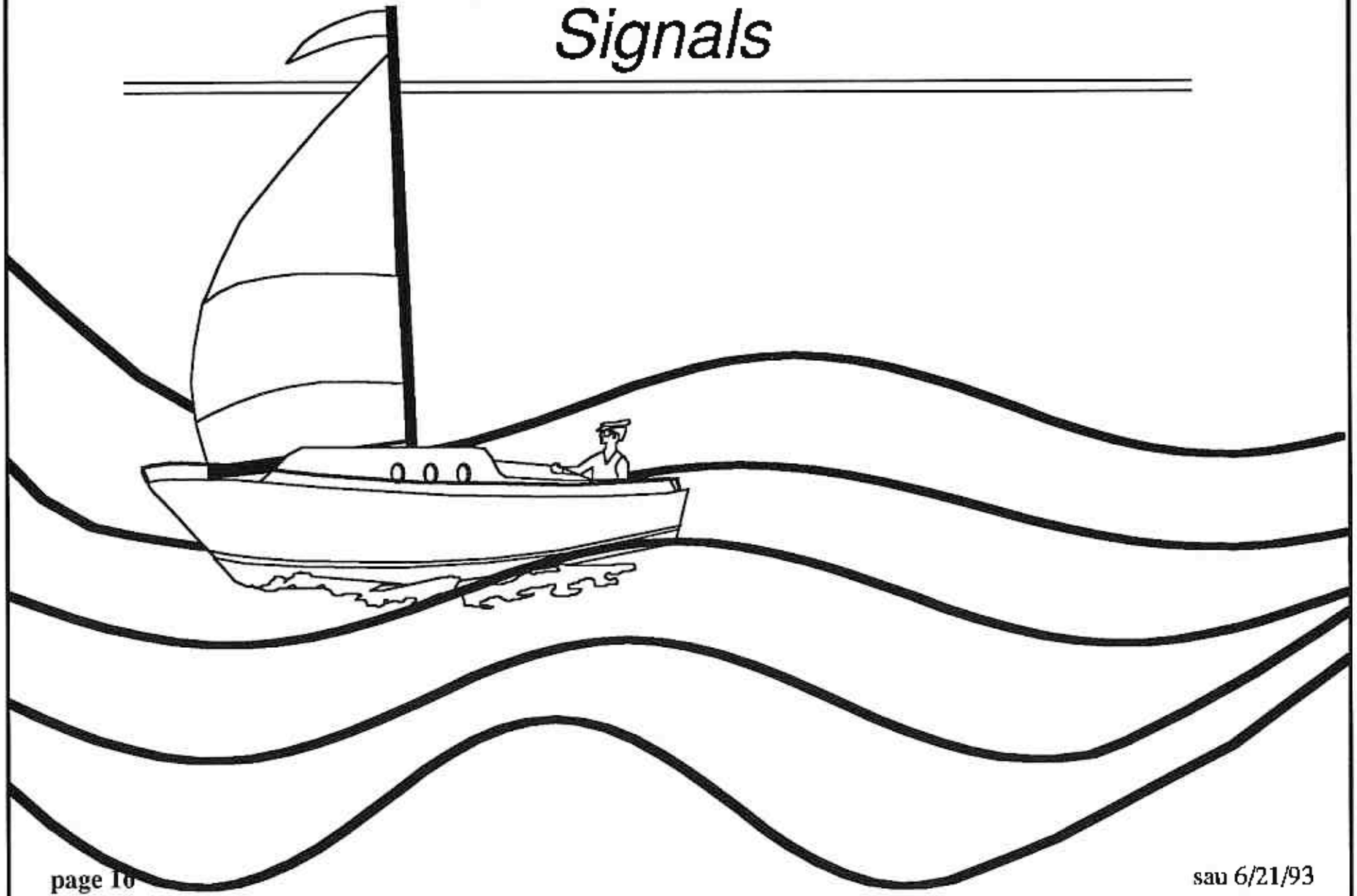
Anatomy of a Musical Sound

Harmonically related components of a vibrating string



f = fundamental frequency

Practical Uses for Sinusoidal Signals



Sine Wave Generation Using Second Order Recursion

$$a = n\Omega+\theta \qquad b=\Omega$$

$$Y_n = A\cos(n\Omega+\theta) = A\cos(a)$$

$$Y_{n+1} = A\cos([n+1]\Omega+\theta) = A\cos(n\Omega+\theta+\Omega) = A\cos(a+b)$$

$$Y_{n-1} = A\cos([n-1]\Omega+\theta) = A\cos(n\Omega+\theta-\Omega) = A\cos(a-b)$$

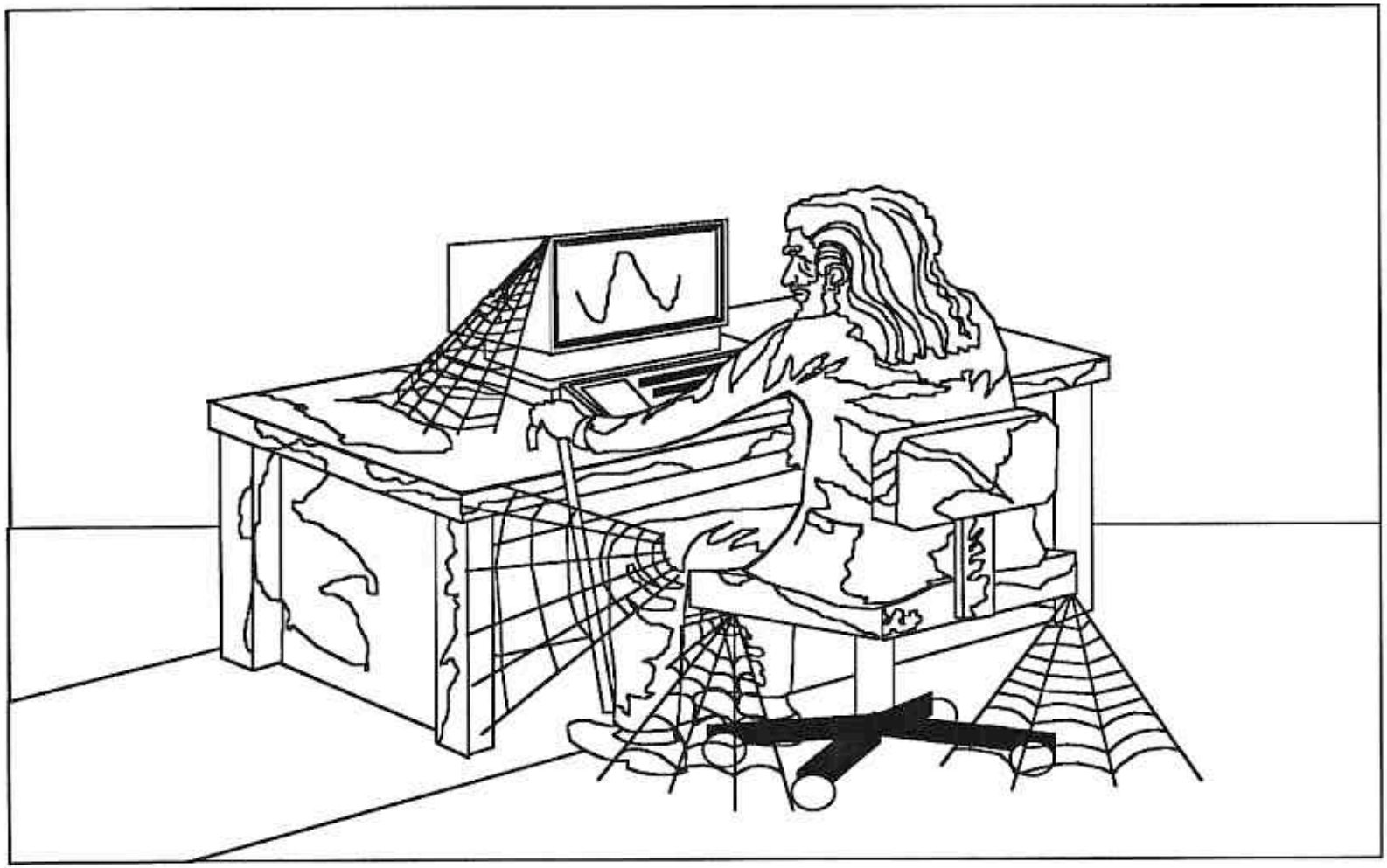
$$\cos(a+b) + \cos(a-b) = 2\cos(a)\cos(b)$$

$$Y_{n+1} = \underbrace{A\cos(a+b) + [A\cos(a-b)}_{} - A\cos(a-b)]$$

$$\underbrace{A\cos(a)}_{Y_n} \quad \underbrace{2\cos(b)}_{2\cos(\Omega)} \qquad -\underbrace{A\cos(a-b)}_{Y_{n-1}}$$

$$\boxed{Y_{n+1} = 2\cos(\Omega)Y_n - Y_{n-1}}$$

Stability of Recursive Sine Wave Generation:
Waiting for the Audio Signal to Drift



Formula for a Sinusoidally Frequency-Modulated Sine Wave

$$Y_n = [\, C_0 + \delta_n \,]\, Y_{n-1} - Y_{n-2}$$

$$\delta_n = C_\delta \delta_{n-1} - \delta_{n-2}$$

$C_0 = 2\cos(\Omega_1)$ Carrier Frequency

$C_\delta = 2\cos(\Omega_2)$ Modulating Frequency

Formula for an Sinusoidally Amplitude-Modulated Sine Wave

$$Y_n = (1+\delta_n)\,[\,C_0\,Y_{n-1} - Y_{n-2}]$$

$$\delta_n = C_\delta\,\delta_{n-1} - \delta_{n-2}$$

$\mathbf{C_0 = 2\cos(\Omega_1)}$ Carrier Frequency

$\mathbf{C_\delta = 2\cos(\Omega_2)}$ Modulating Frequency

Calculating the Frequency of a Sine Wave

$$Y_n = 2\cos(\Omega)Y_{n-1} - Y_{n-2}$$

$$\cos(\Omega) = \frac{Y_n + Y_{n-2}}{2Y_{n-1}}$$

Calculating the Instantaneous Energy in a Sine Wave

$$\cos^2 a - \sin^2 b = \cos(a+b)\cos(a-b)$$

$$\underbrace{A^2 \cos^2(a)}_{Y_n^2} - A^2 \sin^2(b) = \underbrace{A \cos(a+b)}_{Y_{n+1}} \underbrace{A \cos(a-b)}_{Y_{n-1}}$$

$$\underbrace{\boxed{A^2 \sin^2(\Omega) = Y_n^2 - Y_{n+1} Y_{n-1}}}_{\text{Energy}}$$

Sine Wave Generation:
General Second-Order Recursion

$Y_n = r^n A \cos(a)$ r : damping (r<=1)

$Y_{n+1} = r^{(n+1)} A \cos(a+b)$

$Y_{n-1} = r^{(n-1)} A \cos(a-b)$

$$\cos(a+b) + \cos(a-b) = 2\cos(a)\cos(b)$$

$$Y_{n+1} = \underbrace{Ar^{(n+1)}\cos(a+b) + [Ar^{(n+1)}\cos(a-b)}} - Ar^{(n+1)}\cos(a-b)]$$

$$\underbrace{Ar^{(n+1)}\cos(a)}\ 2\cos(b) \qquad -\underbrace{Ar^{(n+1)}\cos(a-b)}$$

$$r\,Y_n \qquad 2\cos(\Omega) \qquad r^2 Y_{n-1}$$

$$\boxed{Y_{n+1} = 2r\cos(\Omega)Y_n - r^2 Y_{n-1}}$$

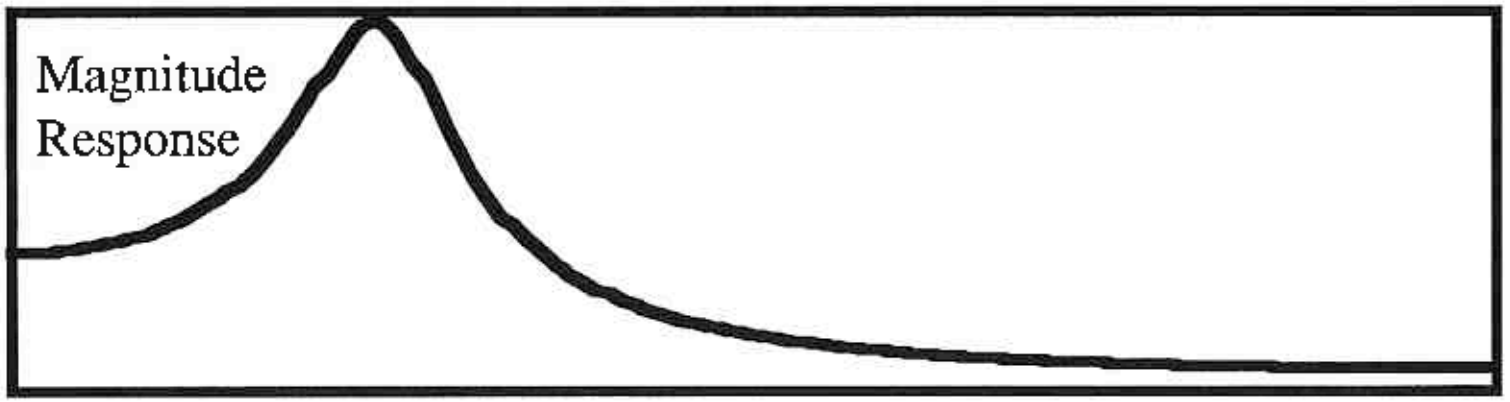
Constructing a Bandpass Filter

$$Y_{n+1} = b_1 Y_n + b_2 Y_{n-1} + X_n$$

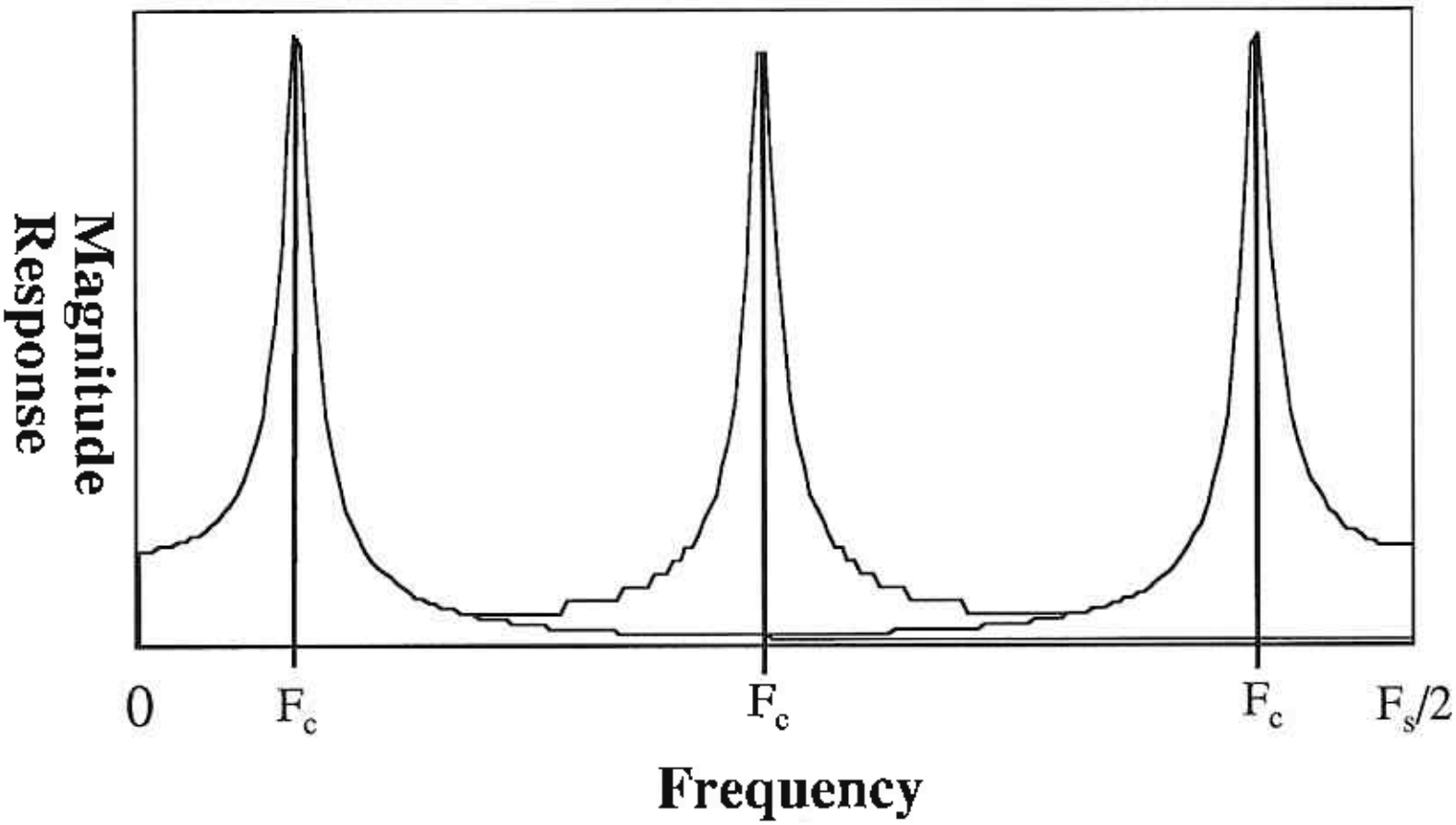
Y: output
X: input

Damped sine wave

Forcing function

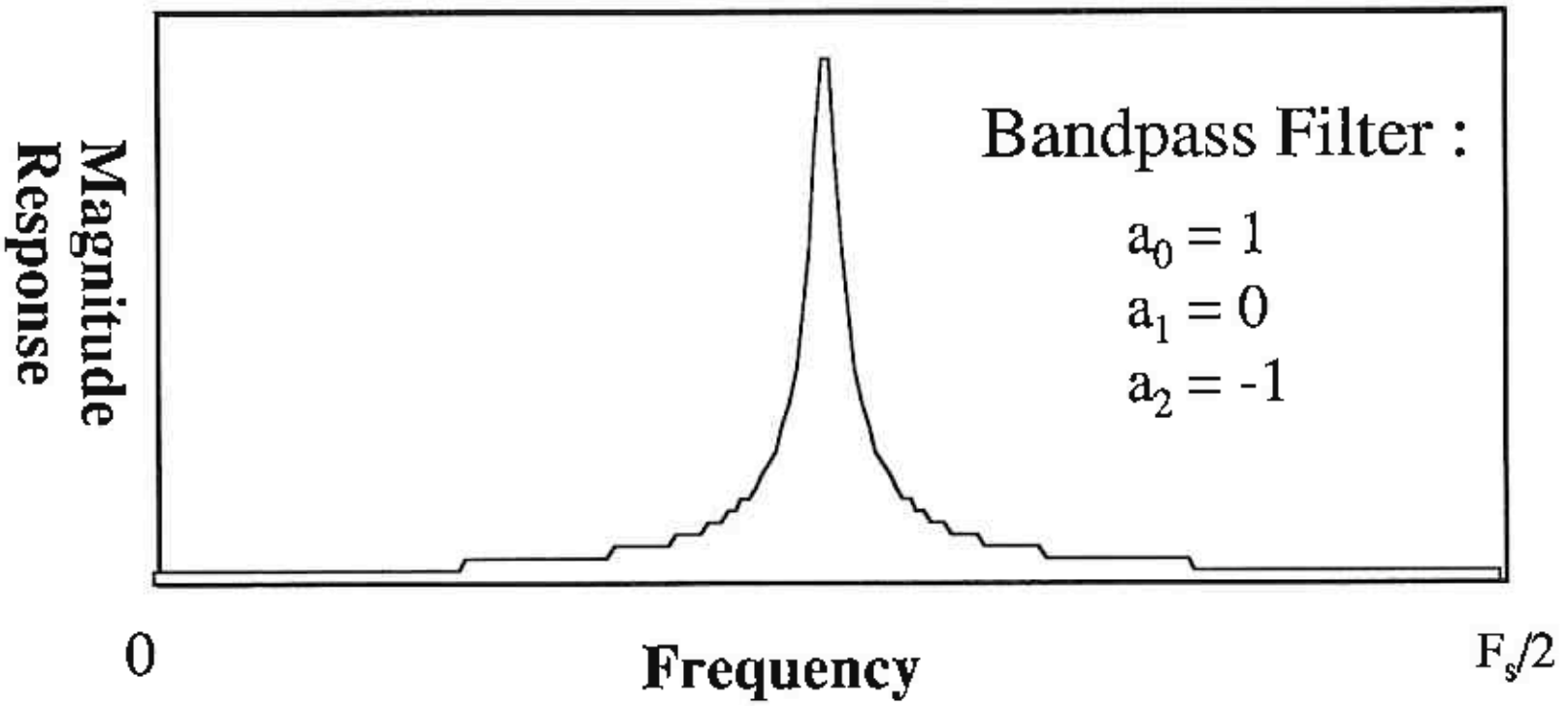


Response of Simple Bandpass Filters

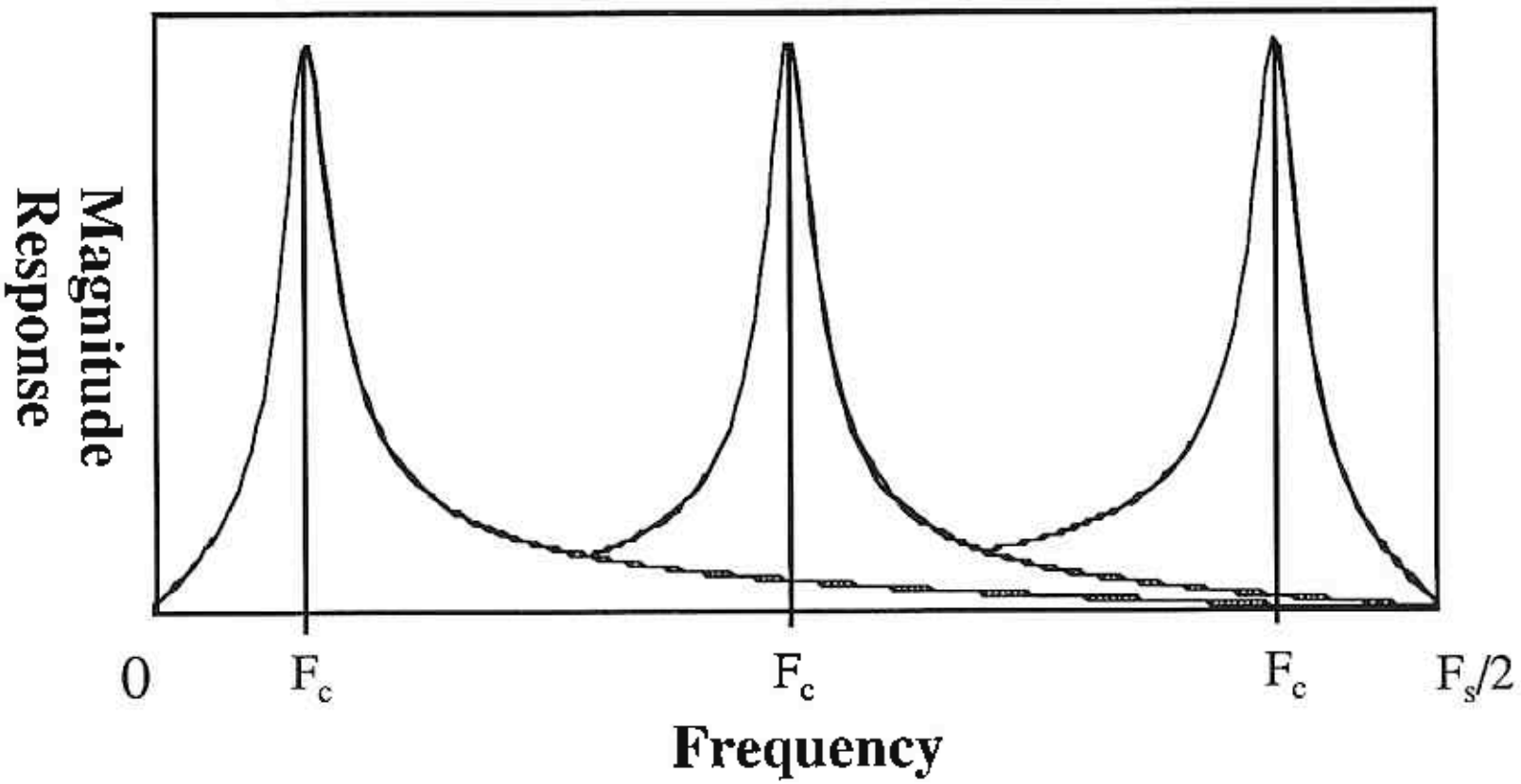


Response of a General Second-order Filter

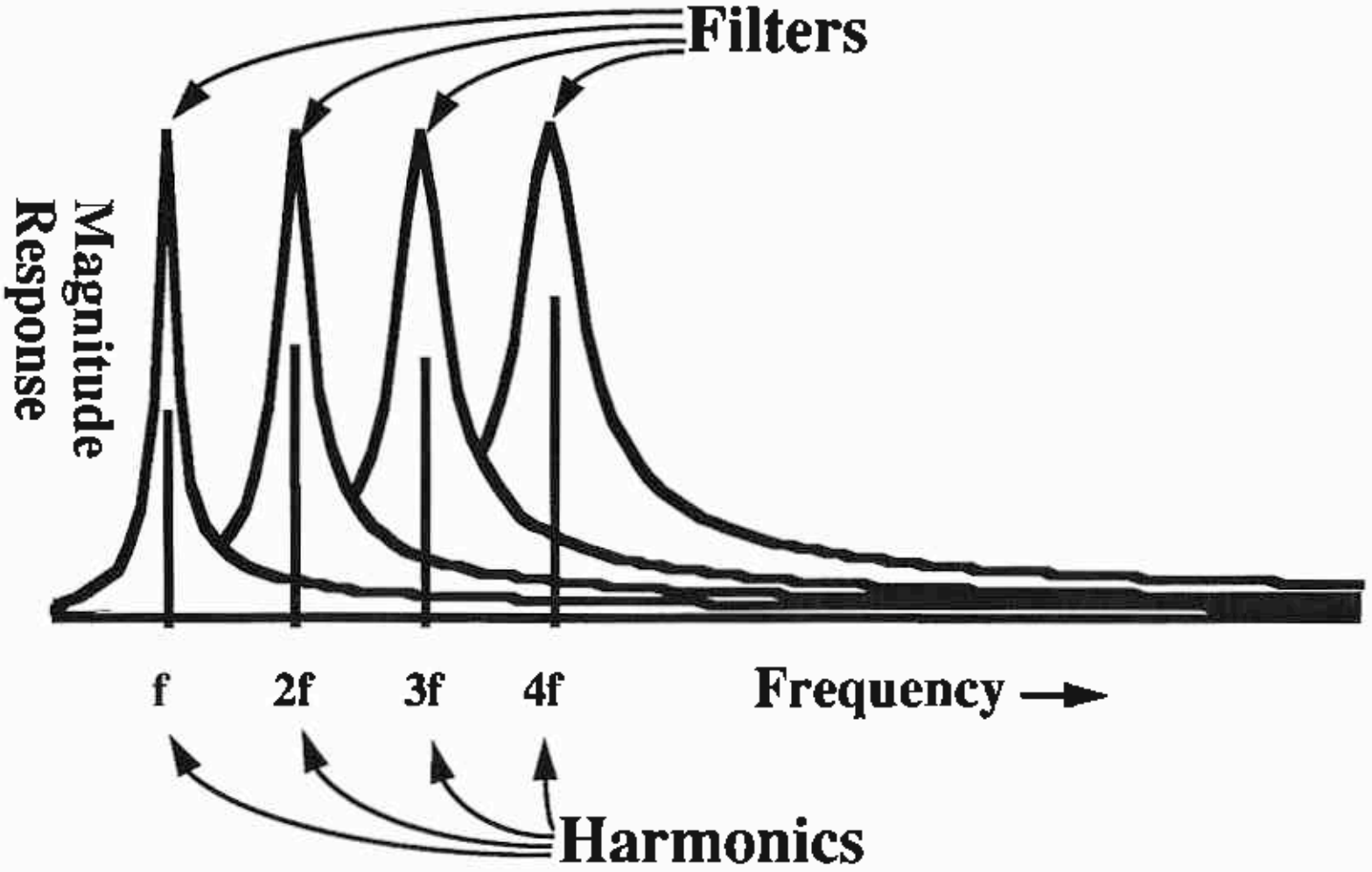
$$Y_{n+1} = a_0 X_{n+1} + a_1 X_n + a_2 X_{n-1} - b_1 Y_n - b_2 Y_{n-1}$$



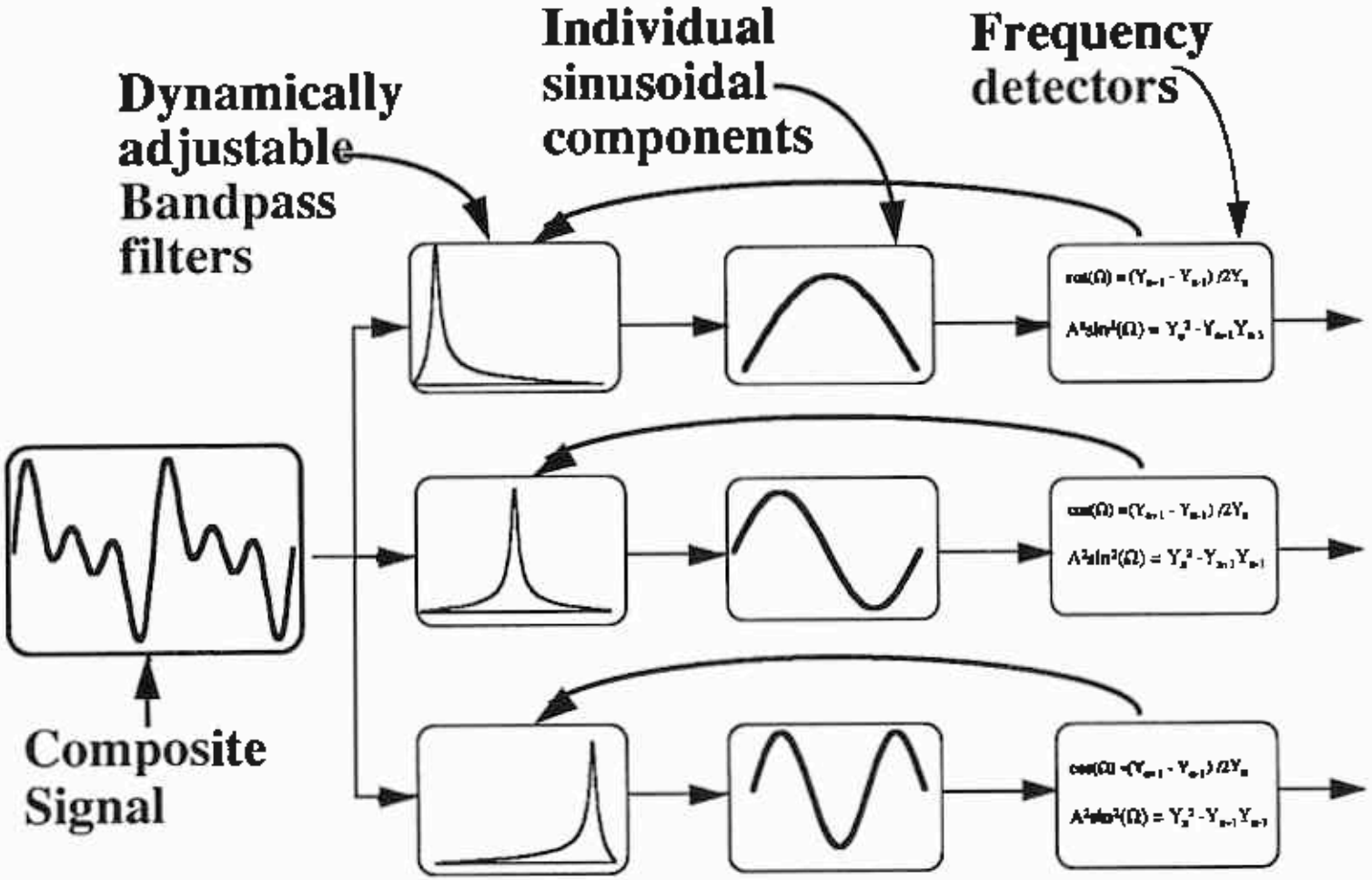
Response of a General Second-order Bandpass Filter



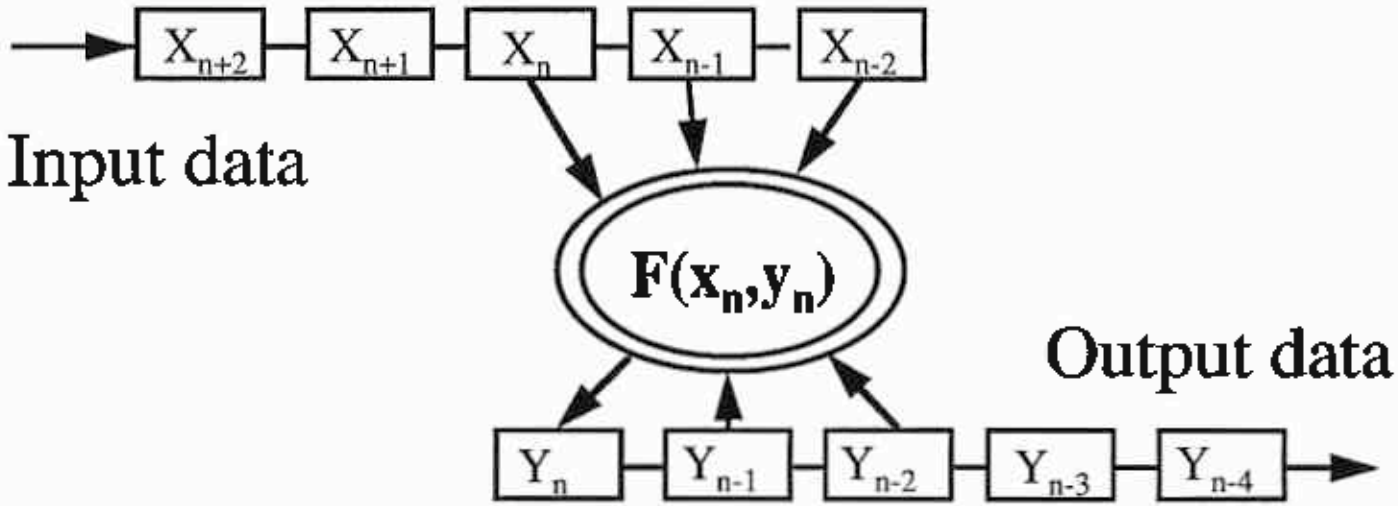
Applying Bandpass Filters to Decompose A Complex Signal



Analyzing Composite Musical Signals



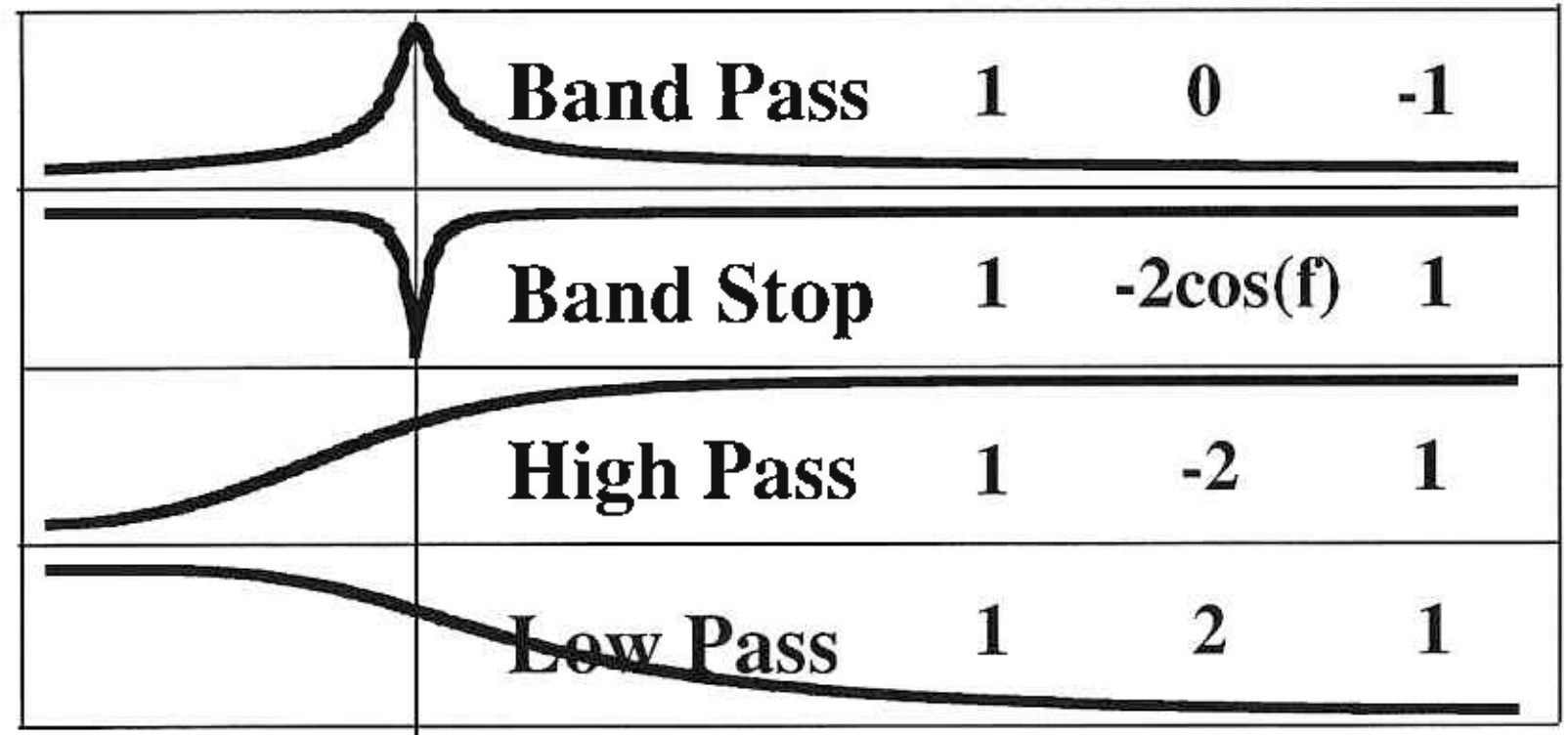
Recursive Implementation Strategy for a Filter



$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} - (b_1 y_{n-1} + b_2 y_{n-2})$$

Some Useful Second-Order Filter Coefficients

	a0	a1	a2
Band Pass	1	0	-1
Band Stop	1	-2cos(f)	1
High Pass	1	-2	1
Low Pass	1	2	1



How Not to Generate a Sine Wave

```
#define RATE        8000.0  /* sampling rate (HZ) */
#define PIE         3.1416

do_sine(freq,amp,buff,dur)
float freq;   /* frequency of generated sine wave */
float amp;    /* amplitude of generated sine wave */
float *buff;  /* where to put the output */
int dur;      /* duration in samples */
        {
        float omega = 2.0 * PIE * freq / RATE;
        int n; /* sample number */

        for(n=0;n<dur;n++) {
                *buff++ = amp * sin(n*omega);
                }
        }
```

Recursive Sine Wave Generation

```
#define RATE    8000.0  /* sampling rate (HZ) */
#define PIE     3.1416

do_sine(freq,amp,buff,dur)
float freq;       /* frequency of generated sine wave */
float amp;        /* amplitude of generated sine wave */
float *buff;      /* where to put the output */
int dur;          /* duration in samples */
    {
    float omega = 2.0 * PIE * freq / RATE;
    float y0 = 0.0;                  /* initial value */
    float y1 = amp * sin(omega);     /* second value */
    float c = 2.0 * cos(omega);      /* recursion coefficient */
    float y;                         /* next output sample */
    float *end = buff + dur;         /* end point */

    while (buff < end) {
        *buff++ = y0;
        y = c * y1 - y0;
        y0 = y1, y1 = y;
        }
    }
```

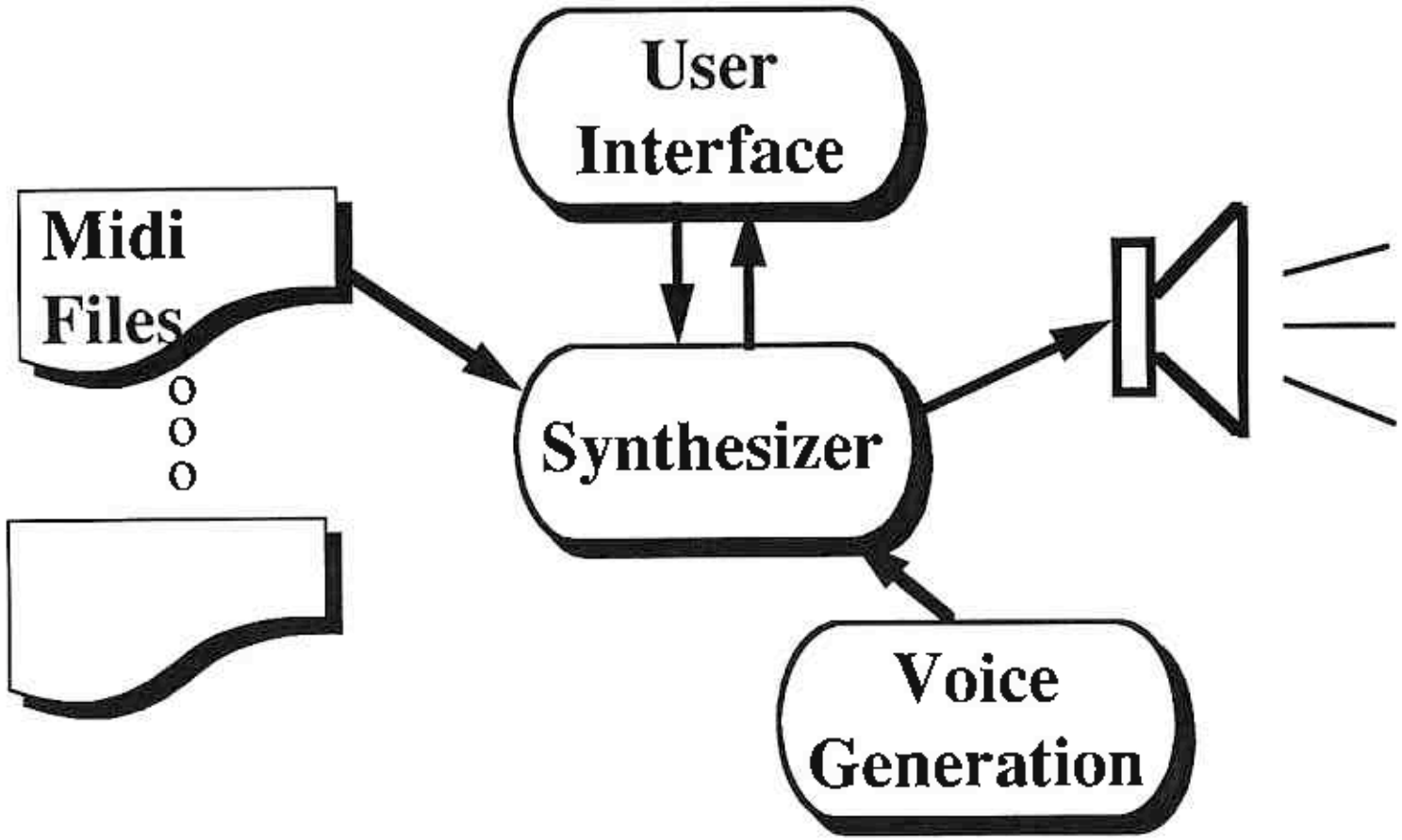
Efficient Sine Wave Generation

```
#define RATE    8000.0 /* sampling rate (HZ) */
#define PIE     3.1416

do_sine(freq,amp,buff,dur)
float freq;       /* frequency of generated sine wave */
float amp;        /* amplitude of generated sine wave */
float *buff;      /* where to put the output */
int dur;          /* duration in samples (must be even) */
    {
    float omega = 2.0 * PIE * freq / RATE;
    float y0 = 0.0;                  /* initial value */
    float y1 = amp * sin(omega); /* second value */
    float c = 2.0 * cos(omega);  /* recursion coeff. */
    float *end = buff + dur;

    while (buff < end) {
        *buff ++ = y0, *buff++ = y1;
        y0 = c * y1 - y0;
        y1 = c * y0 - y1;
        }
    }
```

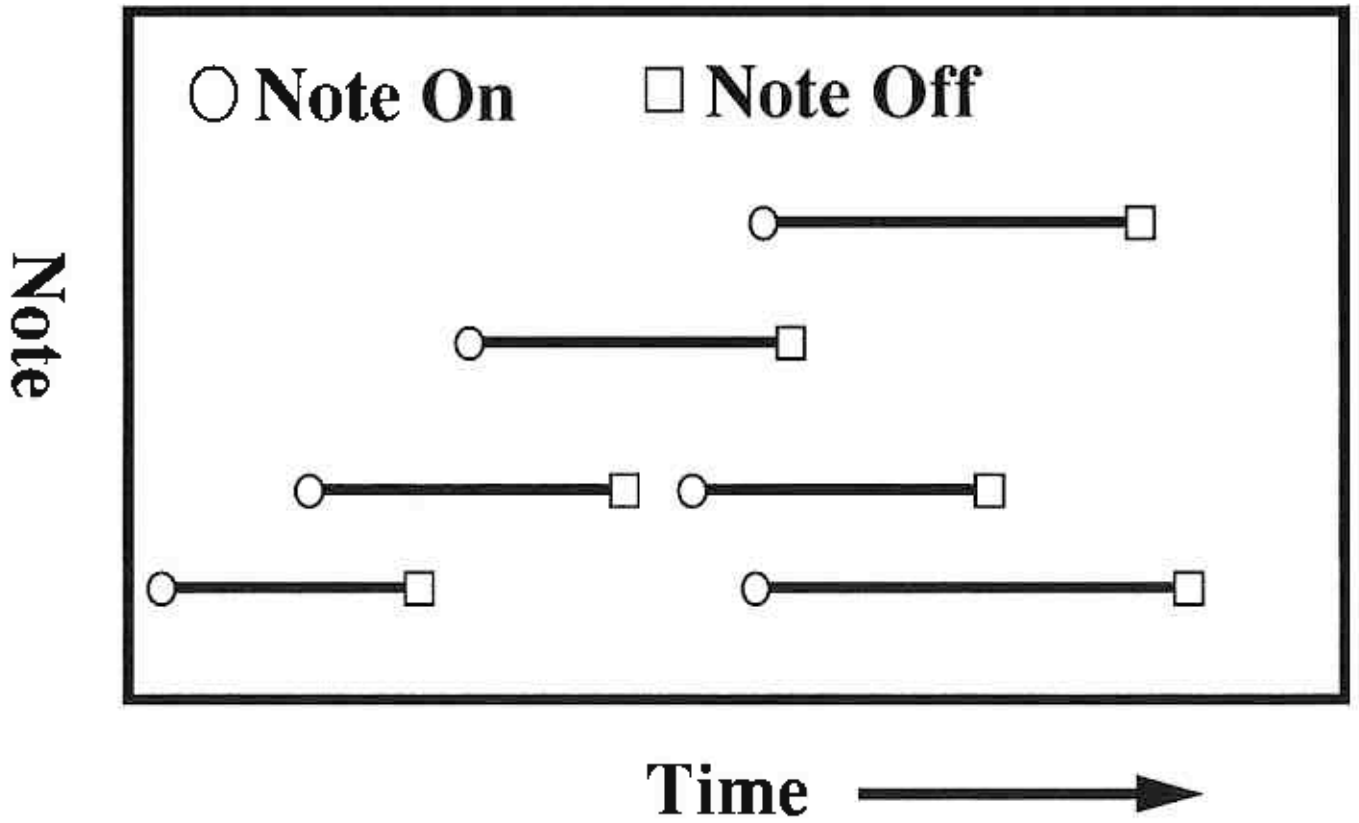
Bandpass Filter implementation

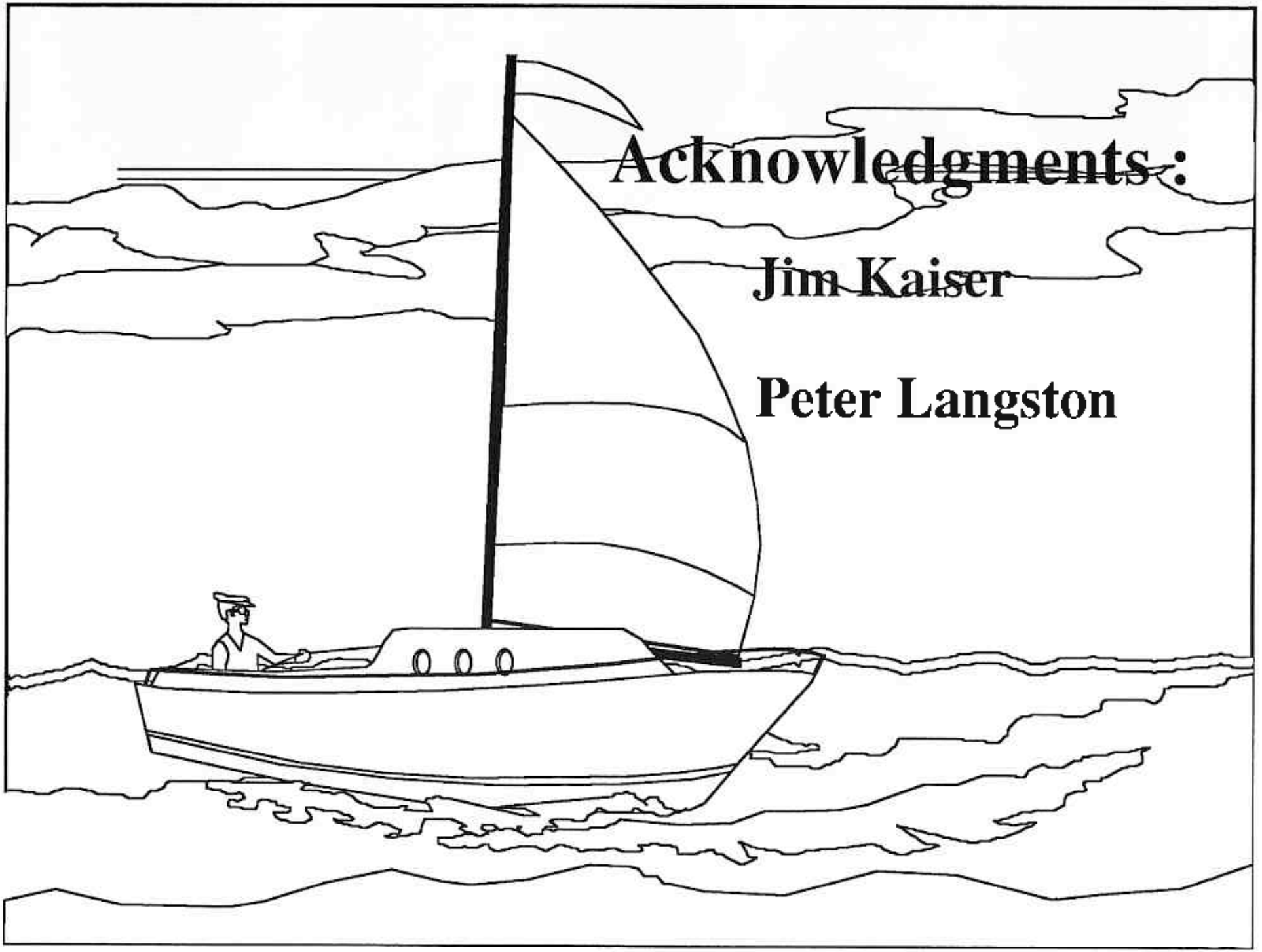
```
#define RATE    8000.0  /* sampling rate (HZ) */
#define PIE     3.1416

do_bpf(freq, damp, in, out, count)
float freq;     /* filter center frequency */
float damp;     /* damping factor */
float *in;      /* input samples (with space for -1, and -2) */
float *out;     /* output samples (with space for -1, and -2) */
int count;      /* number of samples */
     {
     float b1 = 2.0 * damp * cos(2.0 * PIE * freq / RATE);
     float b2 = - damp * damp;
     float *end = in + count;

     while(in < end) {
        out[0] = (in[0] - in[-2]) - (b1*out[-1] + b2*out[-2]);
        out++, in++;
        }
     }
```

A MIDI Synthesizer Block Diagram



MIDI Graphical Respresentation





Acknowledgments :

Jim Kaiser

Peter Langston