CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS
FEBRUARY1987

Department of Music
Report No. STAN-M-55

AN INTRODUCTION TO THE PHASE VOCODER

John Gordon and John Strawn

Research sponsored by

The System Development Foundation

CCRMA
DEPARTMENT OF MUSIC
Stanford University
Stanford, California 94305-8180

An Introduction to the Phase Vocoder

John Gordon and John Strawn

Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford California 94305

1. Introduction

It is often desirable to obtain a time-varying spectral representation of a musical tone. One application of such data (see Moorer 1978, 1979; Holtzman 1980 p. 167) allows the composer to *independently* modify the frequency, time scale and the spectral content of recorded tones, or even orchestral passages. A second allows composers and psychoacousticians to analyse tones and melodic phrases from traditional musical instruments in order to gain insight into the workings of traditional musical sound (Risset 1966; Freedman 1967, 1968; Beauchamp 1969; Grey 1975; Moorer 1978a; Strawn 1981).

The (digital) phase vocoder is a technique for converting a sampled signal into a time-varying spectral format. It is computationally efficient when implemented using the fast Fourier transform (FFT). If the sound is resynthesized from the analysis data, the output will be virtually identical to the original sound if no parameters are changed. The phase vocoder offers certain advantages over the heterodyne filter (Moorer 1973; see also Moorer 1975) used by Grey (1975) and others for time-varying spectral analysis.

This paper will attempt to present the phase vocoder on an intuitive level, so that a musician interested in the technique can understand the key points. A rigorous treatment of the mathematics involved can be found in the work of Portnoff (1976, 1978), on which this paper is based.

A brief history of the phase vocoder will be followed by a description of the analysis technique. Since the phase vocoder analyses the input into sequences of real and imaginary numbers, we will show how to convert this format into magnitude and frequency functions for each harmonic by using a simplified version of the conversion algorithm developed by Moorer (1978a). Two algorithms are included to provide further conversion, should storage or examination of the data be desired before resynthesis. Finally, a way of resynthesizing the sound sequence from the magnitude/frequency functions will be given.

In addition, we will provide a detailed description of these algorithms in the form of code listings written in SAIL (Reiser 1976; Wilcox et al. 1980), which includes Algol as a subset. These SAIL procedures have been compiled, executed, and rigorously tested. Where possible, the code listings invoke routines included in (IEEE Digital Signal Processing

---

Committee 1979), an important collection of Fortran procedures useful in many signal processing applications. Admittedly, it is stretching the pseudo-Algol of the code listings in this article to call a Fortran procedure directly. However, since the programs in the IEEE book are well-documented, carefully tested, and easily available in print and on tape, it seems reasonable to avoid the messy details of designing the required routines independently. Appendix B discusses some important details in the use of these Fortran routines.

2. Historical Overview

The term *vocoder* was coined from VOice CODER, which was the name of a device designed to reduce the bandwidth needed for satisfactory transmission of speech over phone lines (see Dudley 1939). The idea was to pass the speech signal through a set of contiguous bandpass filters, such that the combined output of these filters at a given point in time would be a rough approximation of the spectrum of the input. In theory, by transmitting a few filter coefficients, a savings could be achieved in terms of the transmission bandwidth required to transmit a given signal.

In practice, a savings was not possible, because too many channels were needed to preserve speech quality. There was an additional problem, in that only the amplitude, or magnitude, of each filter output was being transmitted; phase information was thrown out. Thus, the speech resynthesized from the encoded version was never identical to the input, regardless of the number of channels used.

The *phase vocoder* (Flanagan and Golden 1966), developed as an extension of the original vocoder concept, preserved phase information, allowing the input and output of the system to be identical. Schafer and Rabiner (1973b), Portnoff (1976, 1978), and Holtzman (1980) improved the technique, with the result that the speed was increased (in terms of computation time), while still allowing the synthesized output to be identical to the input. This system is the basis for the implementation of the phase vocoder to be discussed in this article.

3. The Analysis Technique

There are two parts to the phase vocoder: an *analysis* side and a *synthesis* side. As its name implies, the analysis side accepts a digital signal as its input and produces what we will call (among other things) *analysis data*; this is a (time-varying) *parametric representation* of the input signal. If this analysis data is left unmodified and then fed to the synthesis side, the output of the synthesis side will be a signal which is identical to the original signal.

Before delving into the details of how the phase vocoder works, it will be useful to introduce an *intuitive* model of the process. We will then cover the many aspects of the analysis technique, presenting the details in close coordination with the code listings.

3.1. Intuitive Models

3.1.1. The Phase Vocoder as a Bank of Filters

In many of the computer music applications to date, the phase vocoder is used to analyse a short tone, such as a digital recording of a tone from a traditional Western acoustic non-percussive instrument, with a frequency that does not change significantly. Assuming that we know the fundamental frequency of the tone, we can design a *low-pass* filter with a cutoff frequency between the first two harmonics. If the recording is passed through this low-pass filter, the output of the filter will be a signal containing only the first harmonic. Similarly, we can design a *band-pass* filter that would extract any particular harmonic. (We are using the term *harmonic* in the sense given in (Moorer 1977).)

The phase vocoder can be seen as a bank of such filters, as shown in Fig. 1. In a typical music-oriented application, the phase vocoder is set up so that only one harmonic falls into each filter passband, or *channel*. However, the output of each filter (a horizontal row of dots in Fig. 1) is not the harmonic that falls into the channel covered by the filter, as one might expect. Rather, each channel produces a *time-varying parametric representation* of what falls into the corresponding filter passband (a horizonal row of dots in Fig. 1). In Portnoff's implementation, when the tone is to be resynthesized, the analysis data (parametric representation) from the various channels is manipulated (with some fairly clever mathematics) to create a signal which is identical to the input, assuming that the analysis data remains unchanged. The synthesis method (Portnoff 1976), which will not be discussed here, has the advantage of being very efficient computationally, although the details of the implementation are tricky. Another disadvantage is the fact that this parametric data is difficult to relate intuitively to the traditional ways of discussing musical sound.

3.1.2. Time-Varying Spectra

Another way of looking at the phase vocoder is to think of the analysis side as producing an entire set of spectral data (a vertical row of dots in Fig. 1) for a very short segment of the input signal at a given point in time. Working from this approach, we can say that the phase vocoder thus produces a time-varying spectral representation of the signal.

In a typical musical application, this spectral data is examined to see how the respective spectral components vary in amplitude and frequency (Dolson (1983) derives a one-to-one correspondence between the amplitudes and frequencies of the spectral components in the input signal and the amplitudes and frequencies produced by the phase vocoder). Furthermore, for synthesis, this amplitude/frequency data can be used to drive a set of sinusoidal oscillators, one for each harmonic, as is a common practice in electronic music. When these sinusoidal components are added, the result should also be identical (within computational accuracy) to the input. This synthesis approach (not Portnoff's) will be presented later in this article.

3.2. The Bank of Filters

3.2.1. Determining the Number of Channels

In the literature on the digital phase vocoder, the number of channels is called N; we will adopt the same convention throughout this article. N depends on the nature of the input signal. We have already asserted that when the frequency remains more or less constant, it makes sense to have each channel represent one harmonic (other options will be mentioned later). This means that N should be set to the sampling rate `sRate`[1] divided by the fundamental frequency of the tone:

$$N = \frac{\texttt{sRate}}{\text{fundamental}}.$$

3.2.2. Code for the Analysis Algorithm

Before examining phase vocoder theory in more detail, let's take a quick look at Fig. 2, which shows code for the analysis side of the phase vocoder. This analysis procedure is intended to be called from another program which takes care of such matters as keeping track of the input signal (probably stored in a file); determining `sRate`, N, and other items to be introduced later; and receiving the output of the analysis (probably to be stuffed into a file).

Specifically, the signal to be analyzed is stored in array `input` in Fig. 2, dimensioned [0:`nSamps`-1], where `nSamps` is the duration of the input signal in samples. For the purposes of this introductory article, we will assume that all of the input signal is processed in one call to the analysis procedure. In most cases, this is impractical because the input signal is too long to fit comfortably into computer memory along with all of the other information that the analysis procedure generates; thus the analysis procedure may have to be modified accordingly. When the analysis is completed, arrays `realPart` and `imagPart` will be filled with the parametric representation of the input.

3.2.3. Establishing the Bank of Filters

It turns out that the code for the analysis procedure does not have to set up an entire bank of filters such as shown in Fig. 1. Rather, it is possible (due to some *extremely* clever mathematics given in Portnoff's article, which we will skip discussing here) to design just one low-pass filter and then to use it for all of the channels. Due to the magic of the mathematics, the low-pass filter effectively turns into a bandpass filter appropriate for each successive channel.

3.2.4. Requirements on the Low-pass Filter

There are, however, some fairly stringent requirements on the low-pass filter, if the resynthesized signal is to be the same as the input. Specifically, (a) the center sample (or median

---

1. In this article, variable names and non-reserved words from the code listings will be printed in a `typewriter`-like font.

sample) of the impulse response must have the value of 1, and (b) the impulse response must be 0 at sample numbers which are multiples of N. Such an impulse response is shown in Fig. 4. These are precisely the constraints on the impulse response of an *interpolating filter,* which is essentially a low-pass filter (Schafer and Rabiner 1973a; Crochiere and Rabiner 1981).

3.2.5. Designing the Low-pass Filter

Digital filter design is a rather complicated subject that is beyond the scope of this article. But there is a fairly straightforward, standard method of designing filters called *windowing* which is used in Fig. 2 (and elsewhere in this article); we will give a brief outline of the process here.

The basic idea is to start with an *ideal* filter and to modify its impulse response as little as possible so that the filter becomes useable but still has the desired filtering properties. An *ideal* filter would allow all of the frequencies within its passband to pass through unchanged, and would completely suppress all other frequencies; however, to achieve such a filter would require an infinitely long impulse response. The impulse response of such an ideal low-pass filter is given by the formula (Rabiner and Gold 1973, p. 91)

$$\frac{\sin(2\pi f_c i)}{\pi i} \tag{1}$$

where f_c is the cutoff frequency divided by the sampling rate (so f_c will be a real number less than 0.5), and i refers to the sample number. The low-pass filter that we want will have a band*width* of sRate/N, passing all frequencies within $\pm$sRate/$2N$. Hence, its *cutoff* frequency will be sRate/$2N$, so that

$$\begin{aligned} f_c &= \frac{\text{sRate}/2N}{\text{sRate}} \\ &= \frac{1}{2N}. \end{aligned}$$

The formula for the impulse response of the ideal low-pass filter in this application is then

$$N\frac{\sin(\pi i/N)}{\pi i}$$

(The extra factor of N is included for normalization to 1.0 at $i = 0$.) This formula can be found in Fig. 2, embedded in a **For** loop.

3.2.6. Designing The Window

As for the rest of this **For** loop in Fig. 2, we can turn the impulse response into a useable filter by multiplying it by an appropriate *window*, which is a kind of weighting function. The impulse response is shortened because the window is required to be 0 after some number of points. (This property of the window also guarantees that at any point in time we will be looking only at a relatively short section of the input signal).

We now need to determine the appropriate window to use. There are several, but the Kaiser window has been found to be best for our purposes. The Kaiser window allows for a

tradeoff between the width of the transition band (that is, how sharp the transition is from the passband to the stopband) and the amount of attenuation in the stopband; this tradeoff is controlled by a parameter called β (beta in Fig. 2). As β is increased, the transition bandwidth increases, but the stopband suppression improves. We typically set β to 6.8, which gives a stopband suppresion of around 71 dB. (For more information on the Kaiser window as well as other windows, see (Rabiner and Gold 1975).)

In Fig. 2, the array win is filled with values for one-half of the appropriate Kaiser window by the subroutine Kaiser, taken from (IEEE Digital Signal Processing Committee 1979, p. 5.2-16); Fig. 5 shows an example of such a window created by this subroutine. We will not dwell further here on the fine points of calling the Kaiser subroutine (see Appendix B), except to point out that since the window is symmetrical about its middle point, only half of the window need be calculated by the subroutine (as shown in Fig. 5).

However, there is one important item which affects the Kaiser subroutine and several other steps in the analysis — how long will the window be? For reasons discussed below under "Time-Aliasing," the input array is divided up into groups of N samples (thus N turns out to be related to more than just the "number of channels"). Another parameter passed to the analysis procedure in Fig. 2, nGroups, helps specify how many of these groups of N samples are processed at a time in the loop called "analysis loop." In the initialization portion of Fig. 2, winLen (the length of the array win filled by subroutine Kaiser) is set to $N \times$ nGroups $+ 1$. Remember that the window is symmetrical about its middle point; there are thus $N \times$ nGroups points in each of the two "wings" of the impulse response, and the $+1$ allows for the one point in the middle. The length of the total windowed impulse response (called impLen in Fig. 2) is thus $2 \times$ winLen $- 1$ points.

Changing the variable nGroups will ultimately affect the separation between adjacent channels of the phase vocoder; higher values of nGroups will produce a longer filter and thus a sharper separation between neighboring channels. On the other hand, a higher value of nGroups means that the amount of the input signal entering into the analysis becomes longer, which makes the resolution in time less sharp. The object is thus to make nGroups as small as possible without losing frequency resolution. We have been able to hear the difference between the cases in which nGroups is 3 and 4, but we have detected no differences when nGroups is larger than 4, at least for tones from traditional musical instruments. Highly transient sounds may require a larger nGroups in order to maintain sufficient frequency resolution.

Now that the window has *finally* been designed, we can window the ideal low-pass filter in Fig. 2 and place the result into array imp, which will hold the impulse response for what we will call the *prototype low-pass filter* for the rest of the analysis calculation. Notice in Fig. 2 that the same value for a point in the filter is placed into imp[i] and imp[−i], thanks to the symmetrical nature of the window and the impulse response.

3.2.7. Filtering by Convolution

Filtering is often done by *convolution,* which can be described as follows: The filter's impulse response is superimposed on the input sequence, and each sample of the impulse response is then multiplied by the corresponding input sample, to yield a *product sequence* . In Fig. 2, this product sequence will be impLen samples long. All these samples are added together to form *one* sample of the *filtered* signal. (Note that this filtered signal is not yet in the parametric form finally produced by the phase vocoder). The impulse response is now shifted one sample in relation to the input sequence, and a new product sequence is

formed, again summed to yield the next output sample. (This process is explained in more detail in (Moore 1978) and (Smith 1983)).

If the phase vocoder required a separate low–pass or band–pass filter for each channel, then we would first have to obtain N impulse responses. Then, for every input sample, we would form N product sequences (one for each channel), add each sequence together, and obtain N output samples, one for each harmonic at the given point in time. This would be a lot of computation!

3.3. Implementation of the Analysis Process

Alternatively, we can start by forming a single *partial product sequence*, multiplying the input signal (such as shown in Fig. 6a on a point-by-point basis with only the prototype low–pass filter. This partial product sequence (Fig. 6b) can be *multiplied* further on a point-by-point basis by a sinusoid at some appropriate frequency (determined by the channel) to obtain one of the N product sequences. Then we can sum each partial product sequence individually, as before, to obtain the output of the channel in question. This has the advantage of being much more computationally efficient, while still producing the conversion to the desired parametric representation (Portnoff 1976, Eq. 4).

Since the partial product sequence is created from samples that both precede and follow the current input sample, we must tack on some extra samples to both ends of the array `input`. The number of these samples on each end is called `extra` (= `winLen` − 1) in Fig. 2. The standard practice is to set all these points to 0, as is done in Fig. 2 (just after the initialization). Array `X` holds the input signal suitably padded with zeros.

3.3.1. Filtering

We can now look at the meat of Fig. 2, which is the loop labelled "analysis loop." The first step is to filter the input signal in array `X` to form the partial product sequence mentioned above. The first **For** loop inside "analysis loop" fills the array `preAlias` (the significance of this name will become clear in the next sections) with the partial product sequence; the filter in array `win` is used for this purpose. Figure 6a and 6b show this filtering process.

Now we meet up with the mathematical tricks that allow some of the steps outlined in the previous section to be simplified or even skipped altogether. It is possible to form just *one* partial product sequence, and then subject it to *time-aliasing* and *rotation*. If this is done properly, then the Fourier transform can be used to take care of multiplying the partial product sequence by the sinusoids (one for each channel) mentioned earlier in this section, and also the summation process. This is of course a considerable savings. We won't explain why the math allows this (Portnoff 1976, Eq. (6) – (8)), but we will look at each step that's required in detail.

3.3.2. Time-Aliasing

The next step is to "fold" the array `preAlias` onto itself one or more times and to put the result into the array called `alias`. Since the array `preAlias` still holds time-domain samples, this process is called *time-aliasing,* analogous to *aliasing* in the frequency domain.

Again, this step is called for by the mathematics in Portnoff's article, which will not be discussed here.

This is also where the variable nGroups, already mentioned above, plays its major role. The array alias is N samples long; preAlias is $2N \times$ nGroups + 1 samples long. This means that preAlias will be folded into the alias array $2 \times$ nGroups times. The result of time-aliasing the filtered signal in Fig. 6b is shown in Fig. 6c.

3.3.3. Rotation

The next step is to rotate the array alias by some appropriate amount. For clarity, a separate array shift is provided in Fig. 2 to receive the shifted array; however, the shifting could be performed using only the array alias (see Fig. 3). The amount of shift is given by the variable inc, which is defined by iSampNo (the number of the current sample of the input) modulo N. When array alias in Fig. 6c is rotated, the result in the array shift will appear as shown in Fig. 6d.

Again, we will ignore the mathematical reasons given in Portnoff's article for including this step. (We are dealing here with the shifting property of the discrete Fourier transform (DFT), as described in (Rabiner and Gold 1975) pp. 34, 57-58). However, this step helps remove the necessity for multiplying the partial product sequence by a different sinusoid for each channel, as outlined above.

3.3.4. Fourier Transform

The next step in the analysis is to take the discrete Fourier transform (DFT) of the array shift.[2] In fact, it was Portnoff's contribution to twist around the mathematics so that the phase vocoder could be implemented with the fast Fourier transform (FFT). Using the FFT offers great advantages because the FFT is computationally much more efficient than the DFT or the brute-force filtering outlined above. Since the FFT can be used to perform the analysis, the digital phase vocoder becomes very attractive as an analysis/synthesis tool.

(Rabiner and Gold 1975) and (IEEE Digital Signal Processing Committee 1979) both devote an entire chapter to the Fourier transform; thus we will not dwell on the details of Fourier analysis here. We should point out, however, that once an initial value for N is determined (dividing the sampling rate by the fundamental frequency, as explained earlier), it is still desirable to "fine-tune" the value to optimize the speed of the FFT. For example, if the number of frequency bands is highly composite (i.e., if N's prime factors are small numbers), an especially efficient version of the FFT can be used. Our standard practice is thus to use the FFT algorithm developed by Singleton (1968) for Fourier analysis of a signal, the length of which is a highly composite number; code for this algorithm is contained in Section 1.4 of (IEEE Digital Signal Processing Committee 1979).[3] Some other routine in Chapter 1 of (IEEE Digital Signal Processing 1979) might be chosen, depending on such

---

2. We are assuming here that the reader is familiar with the DFT and FFT; for an introduction, see (Moore 1978).

3. See also Appendix B.

local considerations as the amount of available memory. In Fig. 2, subroutine FFT accepts the array `shift` as its input and produces as its output arrays A and B, filled with real and imaginary parts, respectively, of the parametric representation produced by the analysis.

3.3.4.1. Saving the FFT Output

We have almost reached the end of the analysis procedure. In Fig. 2, the procedure **analysis** creates as its output the arrays `realPart` and `imagPart`, dimensioned [0:$N/2$, 0:`nAnalPts`-1]. We will ignore the meaning of `nAnalPts` for the moment, and look instead at the last **For** loop in Fig. 2, where these two arrays are filled. The first point to be mentioned is that the succesive samples from arrays A and B are copied into these **realPart** and **imagPart** arrays: the first sample from A becomes the analysis datum for *channel number* 0 in array `realPart`; the second sample from A becomes the analysis datum for *channel number* 1; and so on. Array B is likewise copied into `imagPart`. One invocation of the FFT routine thus provides a complete set of outputs for all of the channels 0 through $N-1$.

However, since the sound sequence is a sequence of *real* numbers, the FFT will be *conjugate symmetric*. This means that the analysis data in arrays A and B will be duplicated in channels 1 and $N-1$, 2 and $N-2$, and so on. Thus we can throw away half of the analysis points (i. e. the analysis data for channels $(N/2)+1$ through $N-1$) with no loss of information. Note, however, that even though one-half of the information in arrays A and B is thus superfluous, these arrays must still be dimensi ned [0:$N-1$] to satisfy the requirements of the FFT routine used here.

The other important feature of this last **For** loop in Fig. 2 is the scaling of the A and B arrays by $1/(N/2)$. When the Fourier transform of an N-point signal is used as the basis for analysis/synthesis to form an identity system, scaling by $1/N$ is required in one of two places: (1) the analysis points must be scaled by $1/N$; *or* (2) each point of the signal resynthesized on the basis of the analysis data must be scaled by $1/N$. In the FFT routine given in (IEEE Digital Signal Processing 1979), the scaling is performed automatically for the inverse transform (i.e. for the resynthesis from the analysis data; the code in question is the `DO 20` loop on p. 1.4-11 of (IEEE Digital Signal Processing 1979)). However, in the implementation of the phase vocoder given here, this inverse transform will not be called. Thus, this scaling must be done explicitly, and we choose to take care of this in Fig. 2. Furthermore, the correct scaling in this case is not by $1/N$, but rather by $1/(N/2)$, as shown in Fig. 2 — for resynthesis, only $N/2+1$ channels will be added together, so that scaling by $1/N$ in this case would produce a signal at half the expected magnitude.

To summarize: the analysis side of the phase vocoder as formulated by Portnoff requires that the input signal be filtered with some appropriate filter. The filtered sequence is subjected to time-aliasing and shifting. The FFT of the shifted signal produces a series of analysis points, one for each channel.

3.3.5. The Meaning of R

At this juncture we should examine more closely the nature of the data contained in the arrays `realPart` and `imagPart` . For a given channel, the *absolute* frequency of the analysis data will be lost as part of the analysis process; what will remain is information about the *difference* between the absolute frequency and the channel's center frequency. (This will become clearer when we examine Figures 9 and 10). This is why we call the analysis

data produced by the phase vocoder a *parametric* representation. An important point here is that the output of each channel is effectively bandlimited by the analysis process to lie within the frequency range $[-\texttt{sRate}/2N, \texttt{sRate}/2N]$. This is significant because, due to the sampling theorem, the output of a given channel no longer needs to be sampled at the original sampling rate. Theoretically, a sampling rate as low as $\texttt{sRate}/N$ would be adequate. In other words, in storing the outputs of the channels, as many as N samples might be skipped. If some of the points to be stored can be skipped, then they do not need to be calculated at all! Appropriate interpolation during re-synthesis (as will be given in Figures 7 and 11) should recover the intermediate samples.

In practice, the phase vocoder is implemented with the variable R, which specifies the number of points to skip in calculating and storing the analysis data. R is required to be less than or equal to N. However, since the filters aren't ideal in practice, and since the interpolation won't be ideal in practice, R should be somewhat less than N. We have found that $N/2$ is a suitable value for R.

3.3.5.1. iSampNo, nSamps

With this information, we can tie up the remaining loose ends in Fig. 2. Examination of the "analysis loop" will reveal that only every Rth point in the input signal is analyzed; the others are simply skipped, although they play a role in the analysis, thanks to the nature of the filtering process.

If the array `input` is dimensioned [0:nSamps−1], and only every Rth point is analyzed, then the arrays `realPart` and `imagPart` can be dimensioned $[0{:}N/2, 0{:}\texttt{nAnalPts}-1]$, where `nAnalPts` is given by $(\texttt{nSamps}-1)/R+1$. The arrays `realPart` and `imagPart` thus hold analysis data sampled at the reduced sampling rate.

3.3.5.2. Data Reduction and the Phase Vocoder

One final point to consider is the immense increase in data that the analysis could conceivably produce. There are N complex numbers in the output for every input sample analyzed (the input samples are presumed to be real, not complex, numbers). This means a file containing the analysis data could be as large as $2 \times N$ times as big as the input sound file. We have already seen that only half of the analysis points need to be retained, since the FFT is conjugate symmetric. Since every Rth point will be skipped in performing the analysis, the increase in data is equal to $[(2 \times N)/2]/R$, which simplifies to N/R. We have already seen that it is reasonable to set N/R to 2. This doubling of the amount of data is usually acceptable.

3.4. Summary of Analysis

The analysis portion of the phase vocoder thus produces time-varying real and imaginary outputs for each of $N/2+1$ channels. The analysis is performed every R points. In the next section, we will explore how these real and imaginary values can be changed to the musically more useful quantities of amplitude and frequency.

4. Conversion to Amplitude and Frequency

In musical applications, it is useful and sometimes necessary to convert the real and imaginary terms provided by the phase vocoder analysis into intuitively more appealing amplitude and frequency data. This section will be devoted to a discussion of the conversion process in Fig. 7, which is mathematically equivalent to the one formulated by Moorer (1978a), though somewhat more efficient computationally. This conversion procedure accepts as its input the arrays `realPart` and `imagPart` filled by the analysis procedure of Fig. 2. As for the other inputs to this procedure, the reader should recognize N and R, and the rest (along with the output arrays) will be discussed below.

Converting the real and imaginary components to amplitude is accomplished with the standard formula from Fourier analysis:

$$\text{amplitude}(n) = \sqrt{a^2(n) + b^2(n)}, \tag{2}$$

where a and b are the real and imaginary components, respectively, at sample n.

On the surface, the frequency conversion process might appear to be equally simple. Recall that the frequency of a sinusoidal component can be thought of as the derivative of the instantaneous phase of the component. Since in standard Fourier analysis the instantaneous phase θ can be calculated as

$$\theta(n) = \text{atan}\left[\frac{b(n)}{a(n)}\right], \tag{3}$$

it might be possible to calculate the frequency value by taking the derivative of θ with respect to time, yielding

$$\text{frequency} = \frac{d\theta(n)}{dt} = \frac{a(n)\frac{db(n)}{dt} - b(n)\frac{da(n)}{dt}}{a^2(n) + b^2(n)}. \tag{4}$$

4.1. The Meaning of Q

There are several difficulties in using these formulae, however. The first is that both of them are non-band-limited, which means that the bandlimited signals in arrays `realPart` and `imagPart` cannot be plugged directly into these formulae without potentially causing some significant distortion if the resulting data are to be used to resynthesize the original signal. Part of the solution is to interpolate the data in arrays `realPart` and `imagPart` back to the original sampling rate (the alternative would be to apply the analysis system at each point in time, i. e. set $R = 1$ for Fig. 2). Of course this would represent a large increase in the amount of analysis data, which should be avoided if possible. Furthermore, the interpolation must be accomplished using the same sort of low-pass interpolating filter found in the analysis procedure, which is computationally more expensive than linear interpolation.

Our experience has shown a compromise can often be reached without affecting the quality of the phase vocoder synthesis output: perform part of the interpolation with the filtering method, and use linear interpolation for the rest. This is the reason for introducing the parameter Q as one of the inputs to the procedure in Fig. 7. Ideally, we should interpolate from the sample rate `sRate`/R of Fig. 2 back to `sRate` itself; instead, we will interpolate

back to sRate $\times$ Q/R in Fig. 7, where Q *must* divide evenly into R. Our current practice is to set Q so that the interpolated data is at a sampling rate which is about 1/4 or 1/8 of the original sample rate.[4]

The outputs of the conversion procedure, then, are the two arrays **freqInterm** and **magInterm**, dimensioned [0:$N/2$, 0:nIntermPts-1], where nIntermPts is given by (nAnalPts-1)$\times Q+1$ (i.e. the number of analysis points at the intermediate sampling rate sRate$\times Q/R$). Of course, it is possible to force the conversion procedure to interpolate back to the original sampling rate simply by setting $Q = R$.

4.2. Interpolation with a Low-pass filter

If Q is an integer (which is a valid assumption for our purposes), we could perform the interpolation simply by inserting $Q-1$ zero-valued samples between every adjacent pair of samples in the real and imaginary analysis data for a given channel, and then pass this new sequence through an appropriate low-pass filter. We could use the same filter that was described above for Fig. 2, with $f_c = 1/2Q$. However, it is more efficient to design a special *interpolating filter* which takes advantage of the fact that $(Q-1)/Q$ of the input samples are 0.

Again, we use the same $\sin(2\pi f_c i)/\pi i$ function given in Eq. (1), but this time with $f_c = 1/(2Q)$. This function will be windowed with a Kaiser window designed by the same subroutine Kaiser which was called in Fig. 2. The subroutine calls are identical, except that the length of the interpolating filter in Fig. 7 is called intLen. One of the inputs to Fig. 7, nQs, functions like nGroups in Fig. 2, except that no time-aliasing occurs in the conversion procedure; nQs is thus involved only in establishing intLen.

The low-pass filter needs to be calculated only once. After this is completed, Fig. 7 enters a long **For** loop called "channel loop", stepping through each channel. The original real and imaginary data for the current channel are padded with zeroes in the arrays tempReal and tempImag. Each of these arrays is interpolated using the filter in the array interp to produce the arrays intermReal and intermImag containing analysis data for a given channel at the intermediate sampling rate sRate $\times$ Q/R.

4.3. Frequency Conversion

The interpolation has solved the problem of Eq. (2) and (4) not being band-limited processes. Eq. (2) can now be used to convert the data to amplitude, as shown in Fig. 7. There are still other difficulties, however, with Eq. (3). One is that special filters known as *linear-phase band-limited differentiators* must be designed to calculate $db(n)/dt$ and $da(n)/dt$. (The first-order approximation $db(n)/dt \approx \Delta b(n) = b(n) - b(n-1)$ is not adequate.) Another is that phase information is lost, which can cause problems in the resynthesis when there are discontinuities in the input signal or in its first derivative. Here, phase information needs

---

4. For the purposes of this article, we are operating under the assumption that it is more efficient to do the analysis with the largest possible R and then to interpolate by a factor of Q before doing the conversion. However, if N happens to be a power of 2, it may be more efficient to make R smaller (such as in the range of R/Q), thereby eliminating the interpolation step prior to conversion.

to be carefully preserved; otherwise, the discontinuity will be smeared in an unpredictable way. Also, phase information needs to be preserved if one wishes to have the resynthesized output signal identical to the input.

It is more appropriate for our purposes, therefore, to use the first-order approximation $d\theta(n)/dt \approx \Delta\theta(n) = \theta(n) - \theta(n-1)$. This allows phase to be easily reconstructed by simply accumulating $\Delta\theta(n)$. We thus have

$$\Delta\theta(n) = \text{atan}\left[\frac{b(n)}{a(n)}\right] - \text{atan}\left[\frac{b(n-1)}{a(n-1)}\right], \tag{5}$$

which is the algorithm used in Fig. 7. (For a more detailed comparison of conversion-to-frequency algorithms, see Appendix A.) In Fig. 7, the variables `thisReal` and `thisImag` contain $a(n)$ (=`intermReal[n]`) and $b(n)$ (=`intermImag[n]`), respectively. Likewise, `thisTheta` holds $\theta(n)$ and `lastTheta` holds $\theta(n-1)$. Eq. 5 is applied on a channel-by-channel, sample-by-sample basis to produce frequency information. After appropriate values for magnitude and frequency have been calculated for the given channel, the results are stored in the output arrays `magInterm` and `freqInterm` at the intermediate sampling rate `sRate`$\times Q/R$.

The procedure `Atan2` in Fig. 7 calculates the quotient of its two arguments and returns the inverse tangent of the quotient, with the sign of the inverse tangent (and thus quadrant information) determined from the signs of the arguments. The result is stored in array `freqInterm` in Fig. 7. The Sail procedure `Atan2` is not universally available, and so the proper sign for the inverse tangent may have to be explicitly calculated by the conversion software. One last important detail: we assume the initial conditions are set so that $\theta(-1) = 0$; for each pass through the "channel loop" in Fig. 7, `lastTheta` is set accordingly. These initial conditions reflect the assumption that each channel starts at its center frequency.

4.3.1. The Meaning of the Data in `freqInterm`

A few words of explanation should be given concerning the array `freqInterm`. This aspect of the conversion can be tricky to explain and understand. To simplify matters, let us assume here that the analysis output of a given channel represents a single time-varying sinusoidal component of a more or less harmonic signal. Remember that for a given channel, the phase vocoder provides information about the *difference* between the *center frequency* of the channel, and the *frequency of the component* found in that channel. Moreover, *for a given channel* (and this is the potentially confusing part), the data in array `freqInterm` will represent the amount of *change* in the *difference in phase* between adjacent samples of the signal analyzed in that channel.

To make this clearer, let us invoke the analogy of wavetable lookup for generating a (sinusoid) waveform. The increment in wavetable lookup corresponds to the *difference in phase* between successive samples just mentioned. The larger the increment, the higher the frequency produced in wavetable lookup. The nominal increment for generating a signal at exactly the center frequency of the channel is determined from the wavetable length (corresponding to 2π radians), the sampling rate, and the frequency to be generated (Mathews 1969). If the increment is changed between two successive samples to produce some frequency other than the center frequency of the channel, that *amount of change* in the increment would be contained in the array `freqInterm`. The "increment" at a given point in time for a given channel can thus be found by starting with the increment implied by

the center frequency of the channel and adding to it all of the "change in increment" values in `freqInterm`, as will become apparent when we discuss Fig. 11. So long as `freqInterm` contains only 0 for a given channel, then the frequency of that channel will revert to the center frequency of the channel. Also, if `freqInterm` is constant but not 0 over several samples of a given channel, then the frequency of the component in that channel will be constant, but different from the center frequency of the channel.

One final point: for both the "change in increment" values ($\Delta\theta(n)$ in Eq. 5) and the "increment" itself, the units are radians per sample, *but at the intermediate sample rate* `sRate` $\times\ Q/R$. When calculating the values for the `freqInterm` array, it is thus necessary to divide by R/Q (as is done in Fig. 7) to change the units to radians per sample at the original sampling rate `sRate`.

4.3.2. Refinements to the Conversion Process

Some further refinements to the conversion procedure are needed, and they will be outlined here. These require the introduction of the Boolean variables `saved` and `magInvert` in Fig. 7, and the real variables `tempMag`, `lastMag`, and `saveTheta`.

The first step in the conversion process of Fig. 7 is still to load the variables `thisReal` and `thisImag` with values from the arrays `intermReal` and `intermImag`. Then we calculate `tempMag`, the magnitude at the current sample number, according to the formula of Eq. (2).

If `tempMag` is exactly 0.0, then there is no point in taking the inverse tangent of the real and imaginary parts according to Eq. (5); the `Atan2` routine in Fig. 7 simply returns $\pi/2$ when `thisReal` $= 0$, regardless of the value of `thisImag`. If we accept this value, in most cases a large phase jump will be introduced into the array `freqInterm`. Remember that `freqInterm` contains data in radians per sample at the original sampling rate `sRate`, data corresponding to the difference in "increment" between successive samples. If this difference is suddenly slammed to $\pi/2$, a significant jump in phase can occur when the component for that channel is resynthesized; and meanwhile the frequency trace for that component will suddenly skip in frequency (in fact, the component will skip by `sRate`/4). Since the magnitude of the component coming from the channel is 0, it is reasonable to assume that the frequency of the component in that channel has not changed. As shown in Fig. 7, this is implemented by setting the appropriate location in `freqInterm` to 0. In Fig. 11, it will become more apparent why this works.

Another source of spurious glitches in the frequency data for a given channel arises when the amplitude envelope of the channel signal changes sign. Although natural sounds are likely to have only positive envelopes, the analysis filtering can induce a change in sign in the amplitude, especially if there are discontinuities in the signal. To help clarify this problem and its solution, consider Fig. 8. If we assume that $\theta(n)$ (Eq. (3)) is a band-limited signal, then $\Delta\theta(n)$ from Eq. (5) should lie within the bounds $\{-$`sRate`$/2N, +$`sRate`$/2N\}$, as indicated by point A in the figure. Of course, as was mentioned above, the arctangent function isn't band-limited; however, the assumption will hold in general unless the signal's amplitude is discontinuous or approaches 0. For points B and C in Fig. 8, this assumption doesn't hold; in these cases the change of phase is close to the maximum possible (π or $-\pi$). However, points B and C can be thought of as having a very small phase change (represented by points D and A, respectively), but with "negative magnitude." (Points B and C are drawn on the unit circle in Fig. 8 for simplicity; but such behavior usually happens only when the magnitude is much less than 1.0.)

If we leave the signal analyzed by the channel at point B or C, then a "glitch" will be produced in the frequency trace. The solution is to move point C back to point A (or B back to D). To compensate for this change, the sign of the magnitude produced by Eq. (2) must also be changed. This is done in Fig. 7 by bringing `thetaDiff` within the bounds $\{-\pi, \pi\}$, and toggling the Boolean `magInvert`. In effect, point C is not moved, but the *representation* of point C is changed to make things more convenient for the user. When the signal hovering around point C eventually switches back to hovering around within the "normal" bounds (*i. e.* within $[-\texttt{sRate}/2N, +\texttt{sRate}/2N]$), `magInvert` in Fig. 7 is toggled, and the conversion procedure returns to producing "normal" outputs until the next phase shift larger than 90° happens. An added bonus of this method becomes apparent when one is working with artificially generated signals: if an amplitude modulator goes negative, the negative modulator will be followed exactly by the phase vocoder analysis! However, when displaying the amplitude traces from the phase vocoder analysis, in many applications it will be sufficient to display only the absolute magnitude of the amplitude.

4.4. Converting to Hertz and Back

The arrays `freqInterm` and `magInterm` produced by the conversion procedure of Fig. 7 are now ready for resynthesis, which will be presented in Fig. 11, or for modification, to be discussed at the end of this article. Since the early days of computer music, composers and researchers have found it useful to create displays of magnitude .nd frequency functions, the latter expressed in Hertz, not radians per second. It might also make sense to first convert the values in `freqInterm` to Hertz if these values are to be stored on disk and perhaps accessed by other programs. (Conversion of the data in `magInterm` to decibels will not be discussed here).

In Fig. 9, the procedure `toHz` fills the array `outFreq` with frequency values, in Hertz, calculated from the values in the array `freqInterm`. The other inputs N, R and `sRate` should be familiar to the reader by now.

The procedure presented in Fig. 10 unravels the work done by Fig. 9. The array `freqInterm` filled by Fig. 10 should be identical in every respect to the array produced by the conversion procedure of Fig. 7.

Both of the "channel loop" **For** constructs in Figures 9 and 10 should clarify the nature of the data contained in the array `freqInterm`. In Fig. 9, the values in `freqInterm` must be multiplied by the sampling rate and divided by 2π in order to keep the units straight. To this we add the center frequency (in Hertz) for the current channel. The process is reversed in Fig. 10. Thus, it is necessary to program explicitly what we have been claiming all along: that the values in `freqInterm` specify a signal which represents the *difference* between the signal coming out of the channel, and the center frequency for that channel.

5. Synthesis

We have now reached the final stage in the phase vocoder: resynthesizing the signal from the analysis data. Synthesis is performed by the procedure `synthesize` given in Fig. 11. By now, all of the inputs to this procedure should be familiar to the reader, except for `freqMult`, which we will assume to be set to 1 until we reach the discussion of modification, below. The procedure `synthesize` uses the values in `magInterm` and `freqInterm` to fill the array `sound`, dimensioned $[0{:}\texttt{nOSamps}-1]$, where `nOSamps` is equal to $(\texttt{nIntermPts}-1)\times R/Q+1$.

Note that if R does not divide evenly into nSamps (Fig. 2), the array sound may be one or more samples shorter than the original array input of Fig. 2. At the sample rates typically used for digital audio applications, however, this discrepancy should cause no problems.

The first step in synthesis, then, is to recover the analysis data at the original sampling rate. As was discussed before, in most applications straightforward linear interpolation, which is computationally much more efficient than interpolation with filters, will be sufficient. $R/Q-1$ samples must be inserted between adjacent samples in the arrays `freqInterm` and `magInterm`. In the implementation of Fig. 11, we choose to fill two arrays `mag` and `freq` with data at the original sampling rate for only one channel at a time.

After these two arrays have been properly filled, some important initialization must be performed for the current channel. In Fig. 11, `centerFreq` is assigned the center frequency of the current channel, this time in *radians*, not Hertz.

The variable `scale` is a multiplier which must be included because $N/2-1$ of the analysis channels were dropped at the end of the analysis procedure. This means that almost all of these channels represent not one but *two* channels, and for these channels the variable `scale` remains at 1.0. The exceptions are the channels numbered 0 and $N/2$, for which `scale` is set to be 0.5.[5]

For each sample (at the original sampling rate) of the current channel, a phase value is calculated and stored in the variable `phase`; recalling the wavetable lookup analogy, `phase` represents the current location in the wavetable. In the first line under "additive synthesis" in Fig. 11, the true meaning of the data contained in the `freqInterm` and `freq` arrays is again underscored: to shift the data in these arrays to their proper positions in the frequency space, the center frequency must be included. In terms of the analogy of wavetable lookup, the quantity `freq[sampNo] + centerFreq` in Fig. 11 represents an "increment." This "increment" is added to `oldPhase` to produce the value of `phase` which is the "location" in the "wavetable." Looking at this step in the synthesis process from a different point of view, the "location" in a "wavetable" is advanced by `centerFreq`, and then adjusted by `freq[sampNo]`, for every sample. At the bottom of Fig. 11, `phase` is stored in `oldPhase` for the next pass. When the value in the `freq` array is 0, then the "increment" is equal to `centerFreq` alone; this means that the channel produces a signal at its center frequency, as has been repeatedly asserted.

As for `oldPhase`, it must be initialized to $-$`freqMult`$\times$`centerFreq`, as shown in Fig. 11. The reasons for this are intimately connected with the initialization of `lastReal` and `lastImag` to $[1,0]$ in Fig. 7. We assume that for all time before `sampNo` $= 0$, the input signal is such that each analysis channel finds a sinusoidal component with amplitude of 1.0. These sinusoids are arranged such that the component in a given channel reaches 1.0 (corresponding to a phase of 0 for a cosine waveform) at a time corresponding to `sampNo` $= -1$. Recalling the meaning of the data in `freqInterm`, `freqInterm[0]` contains the change in phase necessary to go from 0 to the appropriate phase of the signal in a given channel at time `sampNo` $= 0$. If we model the sinusoidal component as $\cos(\omega + \phi)$, then `freqInterm[0]` represents ϕ. For this reason, `phase` in Fig. 11 should be set to `freqInterm[0]` *without* adding in `centerFreq`. Since `centerFreq` is added during every pass through the "additive synthesis" loop of Fig. 11, `oldPhase` must be initialized so that `centerFreq` is effectively cancelled out for that first sample.

---

5. `scale` could be set to 2 and 1, respectively, but then the outputs of the analysis procedure in Fig. 2 would have to be scaled by $1/N$ rather than $1/(N/2)$.

The rest of the synthesis procedure in Fig. 11 is meant to be straightforward. The value of phase, still in radians, is passed to the Cos routine, and the value returned is multiplied by the current magnitude of the channel, contained in array **mag**.

5.1. Testing the Analysis/Synthesis System

If the phase vocoder is indeed an identity system, when the analysis data are not changed before resynthesis, then the implementation should be tested with some suitable input signal.

Figure 12 shows the response to two input signals of the phase vocoder as implemented in the code listings here. Although we will not discuss the details here, a system can be said to be an identity system if the response to an impulse is likewise an impulse, or something very close.

5.2. Other Synthesis Systems

Converting the phase vocoder analysis outputs to amplitude and frequency form is appealing to the musician and psychoacoustician because it makes it possible to discuss the analysis results in intuitive and accessible form. Some of the other available time-varying spectral analysis systems provide a precedent for this step. However, this is somewhat of an aberration in the history of the phase vocoder. It turns out to be computationally much more efficient to resynthesize the final output signal directly from the analysis data in its real/imaginary form. Schafer and Rabiner (1973) give a "direct" form for doing so. Portnoff (1978, p. 32, Eq. 2.39) derives a generalized synthesis equation which includes that of (Allen 1977) as a special case, and gives a more efficient implementation, again based on the FFT. Holtzman (1980) improved upon Portnoff's synthesis method even further. None of these synthesis systems will be discussed here, as it would exceed the bounds of this article.

6. Applications of the Phase Vocoder

6.1. Compositional Applications

The phase vocoder, by itself and in concert with other signal processing techniques, can be used in a compositional context to provide independent control of amplitude, frequency, and spectrum. Although it has not been extensively explored in such applications to date it promises to remain part of the computer musician's repertory well into the foreseeable future.

6.1.1. Modification of Frequency

The `freqMult` input to the `synthesize` routine of Fig. 11 has not yet been discussed. If `freqMult` is set to 1.0, no frequency modification will be carried out. Other numbers can be used, however: setting `freqMult` to 1.5 will transpose the sound up a fifth, and so on. Note that `freqMult` affects both the data contained in array `freq` as well as the `centerFreq` value calculated in Fig. 11. With `freqMult` not equal to 1.0, *both* the frequency of the

original signal as well as its spectrum are shifted together; alterations in timbre are thus to be expected.

6.1.2. Time-Scale Modification

It is also possible to stretch or compress the sound in time, without altering its frequency or spectrum. This is a modification that has been completely ignored in this paper; however, the process is fairly straightforward. By way of handwave, to make the signal slower, simply interpolate even more points in Fig. 11 (change Q to $2 \times Q$, for example), and make other modifications to the code — array bounds, and so on — as necessary. To speed up the signal, simply "leave out" every nth sample of the interpolated analysis data in the arrays `mag` and `freq` in Fig. 11, and adjust the length of array `sound` accordingly.

The theses by Portnoff and Holtzman mentioned earlier were actually aimed at producing time-scale modifications of speech. Holtzman extended Portnoff's work to allow for *time-varying* time-scale modification (with the analysis data in the real/imaginary form). Holtzman applied his system to the time-scale modification of two music passages: *Etude No. 2 in b* by Fernando Sor, and part of the "Promenade" from *Pictures at an Exhibition* by Moussorgsky. Of particular interest is his observation that "...[d]uring compression, the timbre of the instruments changed slightly. This is due to the fact that the resonating time of the instruments was compressed by the same scale factor as the overall signal" (Holtzman 1980, pp. 167-168). Acceptable but noticeable changes were apparently also produced in the timbres of signals subjected to time-varying time-scale modification (Holtzman 1980, p. 168). Time-varying time-scale modifications have, to our knowledge, not yet been tested with the phase vocoder data in the amplitude/frequency form given here.

6.1.3. Spectral Modification

Since the phase vocoder produces a time-varying spectral representation of the input signal, it would seem reasonable to attempt to filter the input signal by modifying the analysis data. This can in fact be done; Portnoff (1978, pp. 38-43 and 60-65) discusses the limitations. Briefly, not every arbitrary filter can be directly implemented in this manner. If arbitrarily detailed changes in time and/or frequency are to be achieved, the analysis and synthesis filters (in Portnoff's synthesis implementation) must be carefully designed.

6.1.4. Applications with Linear Prediction

Moorer (1978a) suggested that the phase vocoder be combined with the linear predictor (Markel and Gray 1976; Makhoul 1976) to provide independent control of frequency and spectrum. Briefly, the linear predictor is a signal processing technique originally inspired, like the phase vocoder, by research into models of speech. In simplistic terms, the linear predictor will "split" a signal into something like an "excitation" function (such as produced by the human vocal cords, or the motion of a bowed string) and a set of time-varying resonances (such as the human vocal tract, or the body of a violin), or filters. If the original "excitation"/filter pair is combined for synthesis, the output of the linear predictor is identical to the input. Tracy Lind Petersen, Charles Dodge, and James A. Moorer have exploited the compositional possibilities of the linear predictor in the form of a synthesis

technique called *cross-synthesis:* if, for example, the "mouthpiece" of a clarinet as an "excitation" source is grafted onto the "vocal tract" of a speaker, the result can be a talking clarinet.

As Moorer (1978a) has pointed out, the phase vocoder can be teamed with the linear predictor such that the "excitation" signal *alone* can be modified in time or frequency, or both. When this modified "excitation" signal is used for resynthesis with the linear predictor, the resulting signal has been shifted in time and/or frequency, *but* the spectral envelope of the signal has not been changed.

This can, however, have unexpected effects; Moorer (1978a, p. 45) warns: "When pitch changes are made by blindly multiplying all the frequency traces [of the "excitation" function] by a fixed factor, we sometimes get a very strange 'choral' effect. Although this may be useful, it is not what we expected..."

In summary, the phase vocoder (perhaps with the linear predictor) promises to provide powerful compositional tools that point beyond the "chipmunk" effect produced by blindly speeding up or slowing down a signal.

6.2. Applications in Psychoacoustic Research

Since the phase vocoder can produce analysis data in the intuitively appealing form of amplitude and frequency functions, it and its predecessors have already proven useful for the analysis of tones produced by traditional musical instruments. Earlier analysis systems such as the heterodyne filter (Moorer 1973; Beauchamp 1969) suffered from a variety of disadvantages. One major problem with time-varying spectral representations of a signal lies in the enormous amount of data produced by the analysis. A long tradition of research (Risset 1966; Beauchamp 1969; Grey 1975; Strawn 1980; Charbonneau 1981) indicates that methods can be found to reduce this data by several orders of magnitude without any significant modification in the timbre of the signal.

6.3. Other Applications and Related Systems

Channel vocoders have been found useful in suppression of additive noise such as the cockpit noise transmitted with an airplane pilot's voice. A discussion of this work or even a complete list of references is beyond the scope of this article; (Gold, Blankenship and McAulay 1981) provide an interesting introduction.

In his doctoral thesis, Petersen (1980) points out that the traditional channel vocoder by its very nature contradicts the nature of human hearing, especially in light of research into critical bands. In place of the constant-bandwidth filters such as shown in Fig. 1, Petersen discusses in some detail the *constant-Q transform,* which features a frequency-dependent filter bandwidth. When properly formulated, the constant-Q transform and its inverse represent an identity system. Petersen presents some applications, especially in the area of masking, perception of loudness, and noise suppression.

Gish (1978) seems to have developed a time-varying analysis/synthesis system which seems to be better able explicitly to incorporate non-harmonic components. The details of his method remain unpublished, however.

7. Using the Phase Vocoder

The amplitude and frequency plots produced by the phase vocoder cannot always be taken at face value. Figure 12b shows a test signal consisting of a sinusoid at 440 Hz, which we will call the carrier, modulated by another sinusoid at 18 Hz (the modulator). Figure 12d shows the output of the synthesis routine of Fig. 11. Figure 13 shows individual amplitude and frequency traces produces by the analysis of Figures 2, 7, and 9.

With a sampling rate of 4800 Hz and N set to 12, the carrier appears in channel no. 1 (Fig. 13b and 13i). Note that the conversion procedure is able to properly track the amplitude modulator below 0 in Fig. 13b. In testing the implementation of the phase vocoder presented here, we found it necessary, as explained earlier, to retain the sign of the magnitude. For *display* purposes, however, it may be more appropriate in many situations to display simply the absolute value of the magnitude curves.

It must be emphasized that the wild frequency traces in parts of Fig. 13 are merely an artifact of the extremely low amplitudes. When the amplitude for a given channel approaches 0, the frequency trace goes wild.

At first glance, one might expect that only channel no. 1 would produce a non-zero amplitude trace. Due to the vagaries of the Fourier process, however, non-zero amplitudes appear in the other channels. Note that the maximum amplitudes in these other channels are significantly smaller than the amplitude of channel no. 1; the outputs for these channels should be interpreted accordingly.

This does not mean, however, that they are completely insignificant and can be discarded. Moorer (1975) and Gish (1978) point out that the noise-like aberrations in amplitude and frequency traces may well include essential information about rapidly fluctuating or inharmonic components, information which is otherwise not explicitly captured by the phase vocoder. At CCRMA, we are in the process of attempting to determine on the basis of psychoacoustic tests how much of this micro-detail can be omitted from the phase vocoder without any perceptible deterioration of the synthesis output side. Given Grey's results (Grey 1975), a significant amount of data reduction should be possible (Strawn 1981).

If the sound is not harmonic, or it is more like a musical phrase with different pitches, the number of channels may be increased. The use of the phase vocoder for analyzing whole musical phrases is only beginning to be explored (see (Strawn 1981, 1983; Dolson 1983)).

Also, bear in mind that the bandpass filters of Fig. 1 are not ideal. Thus, if a certain partial happens to be right at the edge of one passband, it will probably be passed by the adjacent band as well. Therefore, one may want to make N large so that, in effect, "dummy" channels accumulate this "crosstalk" phenomenon. Of course, one can design the filter's impulse response to have an arbitrarily steep rolloff, but then the amount of computation is increased. Hence we have the common trade-off of accuracy versus computation time.

Conversely, if N is too small, it can happen that two spectral components with significant amplitudes will fall into one channel. In this case, the amplitude trace will show "beats" with a frequency corresponding to the *difference* between the two spectral components in the channel.

We usually force N to be a multiple of 4, just to make sure that the prime factors stay small. However, one could resample the input signal so that the new sampling rate divided by the fundamental frequency (in other words, N) would be a power of 2; then an even more efficient FFT could be used in the phase vocoder (see IEEE Digital Signal Processing Committee 1979).

8. Summary and Conclusion

In this article, we have attempted to provide a general introduction to the phase vocoder, in particular for musical applications, and also to place the phase vocoder into the context of signal processing methods for time-varying spectral analysis. The algorithms presented in the code listings here should enable the brave reader to implement the phase vocoder on a wide variety of computer installations. However, in all fairness, we should not deny that a full understanding of the phase vocoder requires a thorough understanding of the mathematical basis for the technique. The extensive discussions in the theses by Portnoff (1978) and Holtzman (1980) should help smooth the admittedly difficult steps to such an understanding; this is mystical math at its finest.

The phase vocoder is currently the most refined technique for producing a time-varying spectral representation of a musical signal; until further refinements come along, it promises to be a valuable compositional and research tool.

9. Acknowledgments

The phase vocoder was originally implemented at Stanford by James A. Moorer. We are also grateful to him for never failing to answer our many, sometimes confused questions. Furthermore, we would like to thank Winken, Blinken, Nod, and ******* for their careful reading of this manuscript and many useful suggestions.

10. Appendix A. Conversion-to-Frequency Algorithms

Section 4.3 and Fig. 7 show one algorithm for converting the real and imaginary phase vocoder outputs to frequency. Since there are several ways to do this conversion, it is useful to compare their advantages and disadvantages in regard to computational efficiency, accuracy, and ability to provide or preserve certain desired information.

Let us use as a point of departure Eq. (4), reproduced here for convenience:

$$\text{frequency} = \frac{d\theta(n)}{dt} = \frac{a(n)\frac{db(n)}{dt} - b(n)\frac{da(n)}{dt}}{a^2(n) + b^2(n)} . \tag{4}$$

This equation implies that the continuous function $d\theta(t)/dt$ is first obtained and then sampled to yield frequency(n). To realize this formula in a computer, we would have to start with a sampled-data function (whether it be $\theta(n)$ or a formula involving $a(n)$ and $b(n)$) and design a linear-phase digital differentiator to calculate $d\theta(n)/dt$ with some desired accuracy. We have not determined how high an order is necessary to obtain proper accuracy, but clearly a first-order differentiator is not sufficient. There is also the possibility of getting huge spikes when the magnitude approaches 0.

In terms of computation, the number of multiplies for the differentiator is 1 plus its order. If $\theta(n)$ is first obtained using Eq. (3) and then differentiated, there will also be the calculation of the arctangent function (which is on the order of seven multiplies at Stanford). If the right-hand side of Eq. (4) is used, both $a(n)$ *and* $b(n)$ have to be differentiated, not to mention the two multiplies needed to form the numerator and the divide to form the final quotient. (The denominator is presumably available from the magnitude calculation.) These calculations need to be done for every sample n.

As was mentioned in the text, however, the real problem with using this algorithm for our purposes is not in the computational problems so much as in the loss of phase information (or the difficulty in reconstructing phase). Why is phase so important? The main reason has to do with resynthesis. Practically every implementation of sine-summation synthesis (whether in hardware or software) is done by accumulating frequency into a phase angle term, which is used in turn to address a sine table. Thus, especially if an identity is to be maintained between input and output signals, the original phase for each sample (and for each harmonic) must be reconstructed. If Eq. (4) is used to compute frequency, it might be necessary to design a digital integrator to recover the instantaneous phase.

Even if the end goal is merely analysis, there is no need to compute a precise value for instantaneous frequency (which Eq. (4) sets out to do). Indeed, "instantaneous frequency" is a meaningless phrase for musical purposes, since the ear cannot ascertain frequency information instantaneously. The most straightforward approach, then, is to approximate frequency by computing "frequency-like" values that can be used to reconstruct phase easily. This leads to the formula $\Delta\theta(n) = \theta(n) - \theta(n-1)$, which is just the inverse of the standard sine-summation synthesis implementation.

If Eq. (5) is used to compute $\Delta\theta(n)$, we still have the seven or so multiplies it takes to compute the arctangent function, but this is all the calculation necessary. The arctangent routine has to be called only once per sample in Eq. (5), since $\theta(n)$ can be saved one pass and used to calculate $\Delta\theta(n+1)$.

The algorithm developed by Moorer (1978a) also computes $\Delta\theta(n)$, and is mathematically equivalent to Eq. (5), though a bit more complicated. There are four multiplies (and two additions) involved in Moorer's formula besides those necessary to compute the arctangent function, and none of the partial products can be saved from one pass to the next. It is possible that Moorer's algorithm is numerically more stable than Eq. (5), though we have found both formulae to give identical results in our tests at Stanford (36-bit floating-point arithmetic).

11. Appendix B. Using the IEEE Routines

This Appendix is included for those planning to use the Kaiser and FFT Fortran routines from (IEEE Digital Signal Processing Committee 1979) included here in Figures 2 and 7. The casual reader should skip this Appendix; indeed, this Appendix will probably be incomprehensible unless the reader has direct access to a copy of the IEEE book.

For the Kaiser routine, the following code is sufficient (IEEE Digital Signal Processing Committee 1979, p. 5.2-16):

Subroutine Kaiser;
Function Ino.

This code can be lifted from the IEEE book without modification.

Using the FFT routine is slightly more complicated. In the following list, the page numbers are taken from the IEEE book:

Subroutine FFT, pp. 1.4-10 through 1.4-11;
Subroutine FFTMX, pp. 1.4-11 through 1.4-15;
Function ISTKGT, p. 1.4-18;
Subroutine ISTKRL, p. 1.4-18; and

the BLOCK DATA initialization on p. 1.4-9.

Note that none of the initialization routines in the "Standards" section of (IEEE Digital Signal Processing 1979) need to be included.

For code given in this article, we found it necessary to use an interface routine (written by Julius O. Smith III) for the actual Sail call to the Fortran subroutines. The interface routine, written in PDP-10 assembly language, unravels the arguments passed by Sail and packages them in a form which can be accepted by the Fortran subroutines.

Furthermore, the Sail compiler produces object code which requires a *Sail*-specific I/O environment. The standard DEC Fortran compiler will include the *Fortran* I/O environment if any Fortran I/O statements are included in the Fortran source code. As we found out, this includes even the STOP command in the error message in Subroutine ISTKGT. Other amusing problems like these will probably crop up at other installations when loading these procedures into code created by compilers other than Fortran.

12. References

Beauchamp, James W. 1969. "A Computer System for Time-Variant Harmonic Analysis and Synthesis of Musical Tones." In H. von Foerster and J. W. Beauchamp, eds. 1969. *Music by Computers*. New York: Wiley, pp. 19-62.

Charbonneau, G. R. 1981. "Timbre and the Perceptual Effects of Three Types of Data Reduction." *Computer Music Journal* 5(2):10 - 19.

Crochiere, R. E., and L. R. Rabiner. 1981. "Interpolation and Decimation of Digital Signals – A Tutorial Review." *Proceedings of the IEEE* 69(1):300-331.

Dolson, Mark Barry. 1983. *A Tracking Phase Vocoder and its use in the Analysis of Ensemble Sounds*. Ph. D. Dissertation. Pasadena, California: California Institute of Technology.

Dudley, H. 1939. "The Vocoder." *Bell Labs Record* 17:122–126.

Flanagan, J. L., and R. M. Golden. 1966. "Phase Vocoder." *Bell System Technical Journal* 45:1493–1509.

Freedman, M. D. 1967. "Analysis of Musical Instrument Tones." *Journal of the Acoustical Society of America* 41:793-806.

Freedman, M. D. 1968. "A Method for Analysing Musical Tones." *Journal of the Audio Engineering Society* 16:419-425.

Gish, W. C. 1978. "Analysis and Synthesis of Musical Instrument Tones." Audio Engineering Society, 61st Convention, New York, Preprint No. 1410(J-3).

Gold, B., P. E. Blankenship, and R. J. McAulay. 1981. "New Applications of Channel Vocoders." *IEEE Proceedings on Acoustics, Speech, and Signal Processing* 29:13-23.

Grey, John M. 1975. "An Exploration of Musical Timbre." Ph. D. Dissertation, Dept. of Psychology, Stanford University. Department of Music Report STAN-M-2.

Holtzman, Samuel. 1980. "Non-Uniform Time-Scale Modification of Speech." M. Sc. and E. E. Dissertation, Department of Electrical Engineering and Computer Science, MIT.

IEEE Digital Signal Processing Committee, IEEE Acoustics, Speech and Signal Processing Society. 1979. *Programs for Digital Signal Processing.* New York: IEEE Press.

Makhoul, John. 1975. "Linear Prediction: a Tutorial Review." *Proceedings of the IEEE* 63:561-580.

Markel, J. D., and A. H. Gray Jr. 1976. *Linear Prediction of Speech.* New York: Springer.

Mathews, Max V., with J. E. Miller, F. R. Moore, J. R. Pierce, and J.-C. Risset. 1969. *The Technology of Computer Music.* Cambridge, Mass.: The MIT Press.

Moore, F. R. 1978. "An Introduction to the Mathematics of Digital Signal Processing." Part I, *Computer Music Journal* 2(1):38-47. Part II, *Computer Music Journal* 2(2):38-60. Reprinted in C. Roads and J. Strawn, eds. 1983. *Computer Music.* Cambridge, Massachusetts: The MIT Press.

Moorer, James A. 1973. *The Heterodyne Filter as a Tool for Analysis of Transient Waveforms.* Report No. STAN-CS-73-379. Stanford University: Department of Music.

Moorer, James A. 1975. "On the Segmentation and Analysis of Continuous Musical Sound by Digital Computer." Doctoral Dissertation, Department of Computer Science, Stanford University. Department of Music Report STAN-M-3.

Moorer, J. A. 1977. "Signal Processing Aspects of Computer Music - A Survey." *Proceedings of the IEEE* 65(8):1108 - 1137. Revised and updated version in C. Roads and J. Strawn, eds. 1983. *Computer Music.* Cambridge, Massachusetts: The MIT Press.

Moorer, James A. 1978a. "The Use of the Phase Vocoder in Computer Music Applications." *Journal of the Audio Engineering Society* 26:42–45.

Moorer, James A. 1978b. "The Use of the Linear Predictor in Computer Music Applications." *Journal of the Audio Engineering Society* 27:134-140.

Petersen, T. L. 1980. "Acoustic Signal Processing in the Context of a Perceptual Model." Ph. D. Dissertation, Computer Science Department, University of Utah.

Portnoff, Michael R. 1976. "Implementation of the Digital Phase Vocoder Using the Fast Fourier Transform." *IEEE Proceedings on Acoustics, Speech, and Signal Processing* 24:243–248.

Portnoff, M. R. 1978. "Time-Scale Modification of Speech based on Short-Time Fourier Analysis." Doctoral Dissertation, Department of Electrical Engineering and Computer Science, MIT.

Rabiner, L. R., and G. Gold. 1975. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, New Jersey: Prentice-Hall.

Reiser, J., ed. 1976. "SAIL." Report STAN-CS-574. Stanford: Stanford University Artificial Intelligence Laboratory Memo.

Risset, Jean-Claude. 1966. *Computer Study of Trumpet Tones.* Murray Hill, N. J.: Bell Telephone Laboratories.

Schafer, R. W., and L. R. Rabiner. 1973a. "A Digital Signal Processing Approach to Interpolation." *Proceedings of the IEEE* 61:692-702.

Schafer, R. W., and L. R. Rabiner. 1973b. "Design and Simulation of a Speech Analysis-Synthesis System Based on Short-time Fourier Analysis." *IEEE Transactions on Audio and Electroacoustics* AU-21:165–174.

Singleton, Richard C. 1968. "An Algol Procedure for the Fast Fourier Transform with Arbitrary Factors." *Communications of the ACM* 11:776–779.

Smith, J. O. III. 1983. "Introduction to Digital Filter Theory." In C. Roads and J. Strawn, eds. 1983. *Computer Music.* Cambridge, Massachusetts: The MIT Press.

Strawn, John M. 1981. "Approximation and Syntactic Analysis of Amplitude and Frequency Functions for Digital Sound Synthesis." *Computer Music Journal* 4(3):3-24.

Strawn, John M. 1983. "Research on Timbre and Musical Contexts at CCRMA." In T. Blum and J. Strawn, ed. *Proceedings* of the 1982 International Computer Music Conference, Venice, Italy. San Francisco, California: Computer Music Association, 1983, pp. *** – ***.

Wilcox, C. R., M. L. Dageforde, and G. A. Jirak. 1980. "Mainsail Language Manual." Report STAN-CS-80-791. Stanford: Computer Science Department.

Captions for Figures

Fig. 1. The phase vocoder analysis (left) works like a set of band-pass filters spaced equally from 0 Hz to one-half of the sampling rate f_s. Each rectangle on the left corresponds to an analysis channel, with one filter per channel. Each channel produces a time-varying parametric representation (real and imaginary parts) of whatever segment of the input signal's spectrum falls within the filter passband. If the analysis data remains unchanged, then it can be used to synthesize a signal $y(t)$ identical to the input signal $x(t)$.

Fig. 4. Impulse response of the prototype lowpass filter (see Section 3.2.4), as calculated by the expression `SIN(pi*i/N)/(pi*i)` in Fig. 2.

Fig. 5. The Kaiser window as returned in array `win` by subroutine `Kaiser` in Fig. 2.

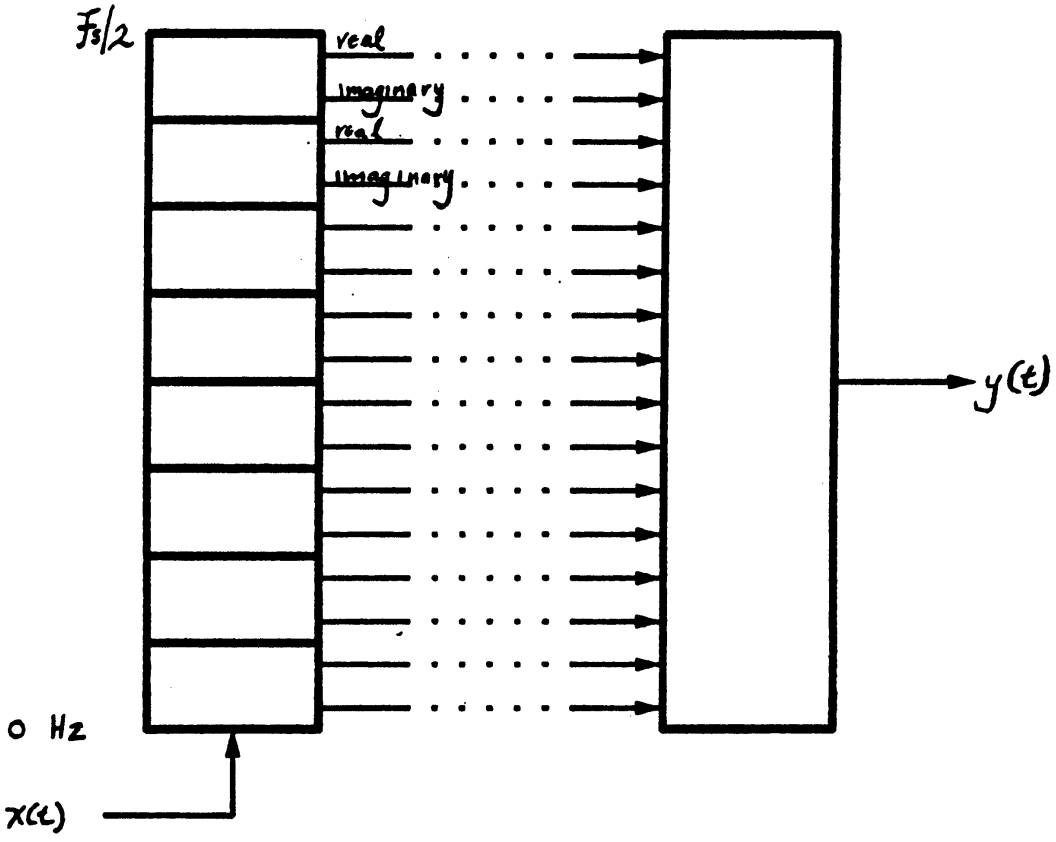
Fig. 6. Effects of the processing in the "`analysis loop`" of Fig. 2. a). An input signal, contained in array `X` of Fig. 2. b). The input signal is multiplied by the windowed low-pass filter contained in array `imp` of Fig. 2; the product shown here is stored in array `preAlias`. c). The signal in Fig. 6b is time-aliased and stored in array `alias`. d). Array `shift` in Fig. 2 holds the signal of Fig. 6c, shifted here by `inc` samples.

Fig. 8. Spurious glitches in the frequency outputs of the phase vocoder can be avoided. The point at $[1, 0]$ (not shown explicitly in the figure) corresponds to the center frequency for the channel. As the frequency analyzed by the channel moves from the center frequency, Point A moves up or down from the x axis along the circle. Point A should lie within the bounds $[-f_s/2N, +f_s/2N]$ shown in the figure. Spurious phase shifts (larger than 90°) are shown at points B and C. In the code of Fig. 7, point B is moved temporarily to point D, and point C is moved to point A to avoid large phase jumps in the output.

Fig. 12. (a) and (b) are test signals input to the phase vocoder. The signal in (a) is a series of positive- and negative-going impulses. (b) is a sine wave with sinusoidal amplitude modulation. (c) and (d) show the output of the phase vocoder for the analysis/synthesis system. Note the small glitches in (c), which we ascribe to numerical inaccuracies. Figure 13 gives the individual amplitude and frequency traces for the sine wave in (b).

Fig. 13. Individual channel outputs for analysis of the signal of Fig. 12b. The amplitude values in the left-hand column are stored in array `magInterm` by Fig. 7. The right-hand column shows the frequency output of Fig. 9, as contained in array `freqInterm`. With $N = 12$ and `sRate`=4800 Hz, $N/2+1 = 7$ channels are shown here, numbered 0 through 6 starting at the top. Note the varying scales for the different plots.

Figure 1



```
procedure analysis(real array input; integer nSamps,N,R,nGroups;
   real array realPart,imagPart);

   begin

   comment
   Inputs:
   array input, dimensioned [0:nSamps-1]
   N            number of phase vocoder channels (see text)
                  --- N assumed to be even
   R            compression ratio --- see text
   nGroups      number of groups of N samples in 1/2 of ImpLen
   nSamps       length of original signal to be analyzed

   Outputs:
   arrays realPart, imagPart, dimensioned [0:N/2, 0:nAnalPts-1]
                      where nAnalPts is the largest integer in (nSamps-1)/R + 1
   ;

   integer
      impLen,   comment length of impulse response of prototype lowpass filter
                        (2 * N * nGroups + 1);
      extra,    comment there are this many extra zeros at each end of array X;
      winLen,   comment Size of positive half of window;
      inc,      comment number of samples to rotate Alias;
      iSampNo,  comment index for input points in array X at original sampling rate;
      oSampNo,  comment index for output points in arrays realPart, imagPart
                        at sampling rate SRate/R;
      i, k;     comment miscellaneous indices;

   real
      beta;     comment parameter for Kaiser window;

   real array X[-N*nGroups : nSamps*N*nGroups-1];
              comment holds input signal surrounded by zeros;
   real array win[0 : nGroups*N];
              comment win holds positive half of window for windowing
                      low-pass filter;
   real array imp,preAlias[-nGroups*N : nGroups*N];
              comment imp holds impulse response of windowed low-pass filter,
                      preAlias holds windowed signal before time-aliasing;
   real array alias, shift[0:N-1];
              comment alias holds windowed signal after time-aliasing,
                      shift holds rotated alias array;
   real array A,B[0:N-1];
              comment A and B will be returned from the FFT with the cosine and
                      sine coefficients for frequencies between 0 and pi;
```

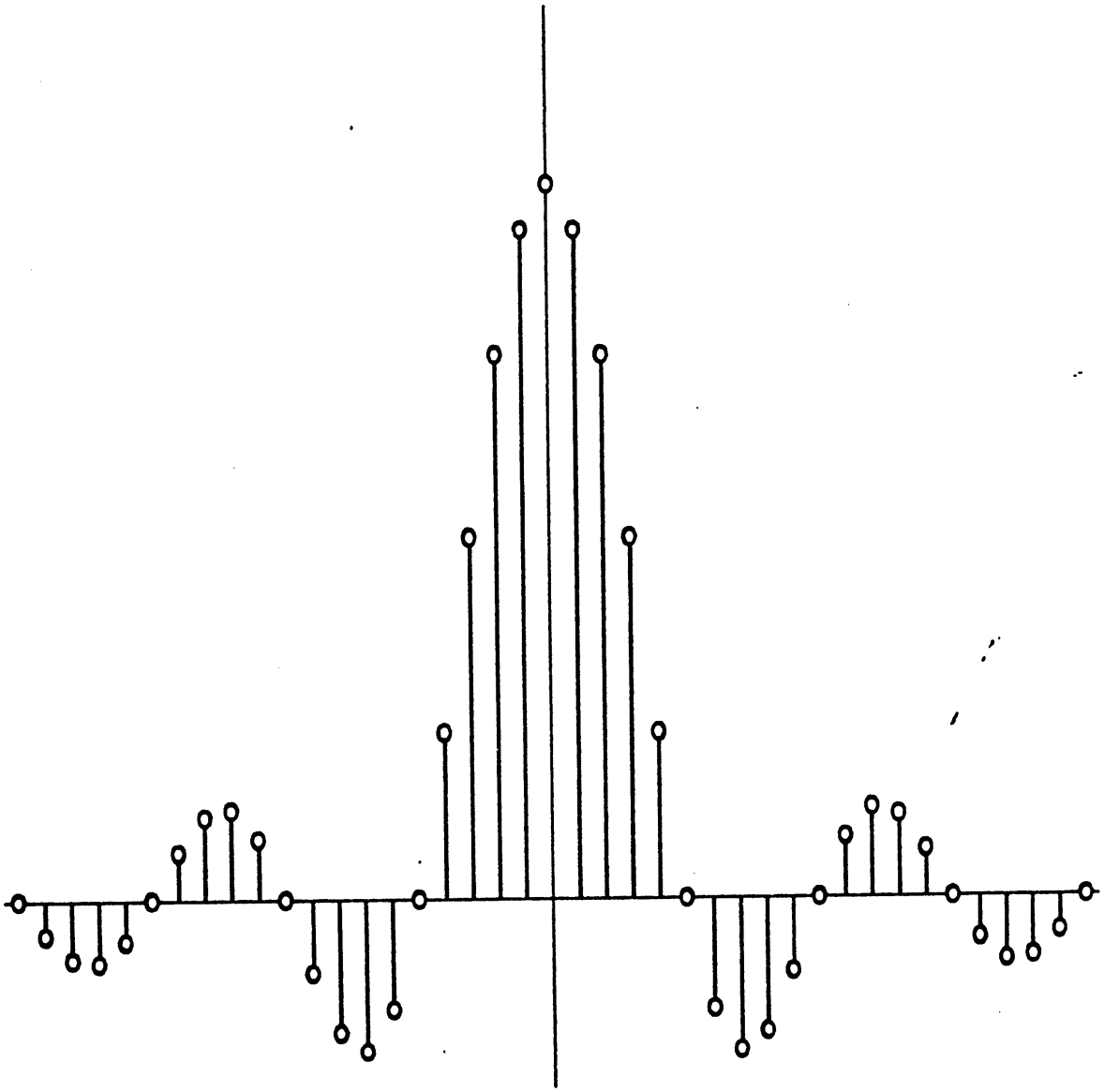
```
comment initialization;
extra  := N*nGroups;
winLen := N*nGroups + 1;
impLen := 2*winLen - 1;
beta   := 6.8; comment This makes stopband for each filter about 71 dB down;

comment stuff array win with half of the symmetric Kaiser window;
Kaiser(impLen,win,winLen,1,beta); comment "1" means that impLen is odd;

comment window impulse response of prototype lowpass filter.  imp is symmetric
        about the 0th (middle) sample;
imp[0] := win[0];
for i := 1 step 1 until N*nGroups do
   imp[i] := imp[-i] := N*win[i]*SIN(pi*i/N)/(pi*i);

comment set up array X;
for i := -extra step 1 until -1 do X[i] := 0;
for i := 0 step 1 until nSamps-1 do X[i] := input[i];
for i := nSamps step 1 until nSamps+extra-1 do X[i] := 0;

oSampNo := 0;
for iSampNo := 0 step R until nSamps-1 do
   begin "analysis loop"

   comment filter input signal;
   for i := -extra step 1 until extra do
      preAlias[i] := X[iSampNo + i] * imp[i];

   comment time-aliasing;
   for i:=0 step 1 until N-1 do
      begin
      alias[i] := 0;
      for k := -nGroups step 1 until nGroups-1 do
         alias[i] := alias[i] + preAlias[k*N + i];
      end;

   comment rotate time-aliased array;
   inc := iSampNo mod N;
   for i := 0 step 1 until inc-1 do
      shift[i] := alias[N + i-inc];
   for i := inc step 1 until N-1 do
      shift[i] := alias[i-inc];

   comment take FFT of the values stored in array shift;
   for i:=0 step 1 until N-1 do
      begin comment set up arrays for FFT;
      A[i]:=shift[i];
      B[i]:=0;
      end;
   FFT(A, B, 1, N, 1, -1);
   comment The "1" and "-1" are explained on p. 143 of (Digital Signal
         Processing Committee 1979) and should not be changed.;
```

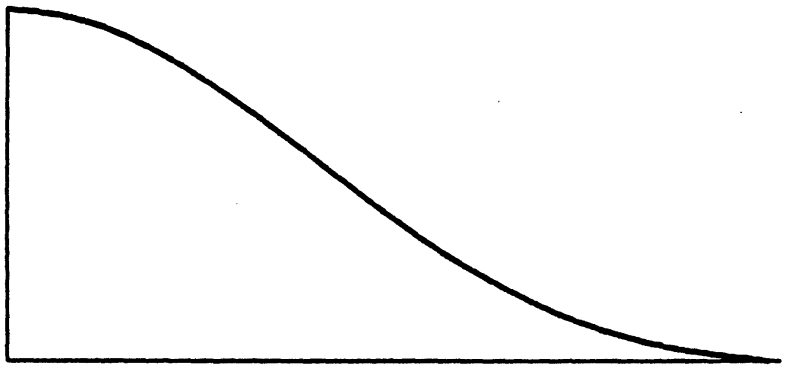
```
      comment For realPart and imagPart, FFT determines N/2+1 values at oSampNo;
      for i := 0 step 1 until N/2 do
         begin comment see text for explanation of scaling;
         realPart[i,oSampNo] := A[i]/(N/2);
         imagPart[i,oSampNo] := B[i]/(N/2);
         end;
      oSampNo := oSampNo+1;
      end "analysis loop";
   end;
```

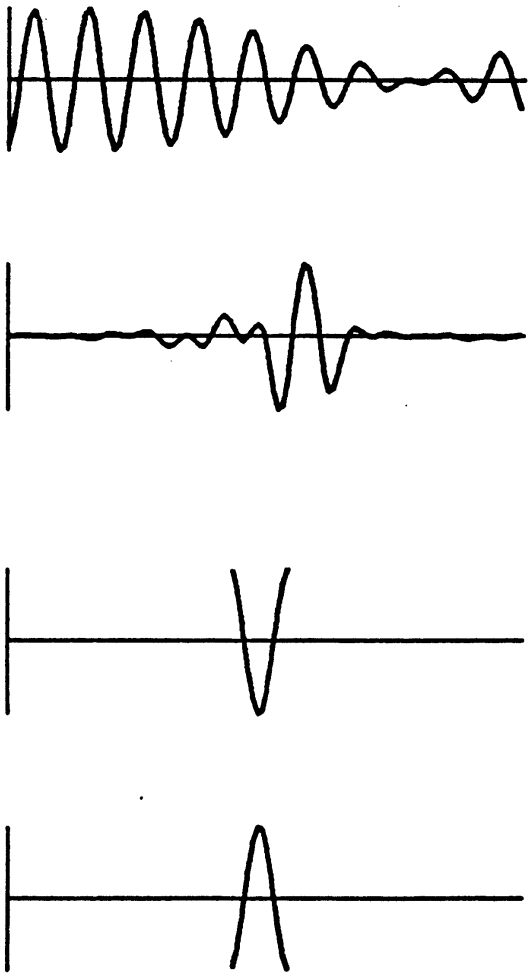
Fig. 2. Phase vocoder analysis of a digital signal using the Fast Fourier Transform (FFT). as given in Portnoff (1976). The algorithms in this article are given in SAIL, although we have attempted to make the code resemble Algol as much as possible. Throughout these code listings, we assume the existence of a globally available real variable **pi** initialized to 3.14159265. The arrays **alias** and **A** are superfluous, as explained in the text, but are included for additional clarity. The routines **Kaiser** and **FFT** are taken from (IEEE Digital Signal Processing Committee 1979). The operator a **mod** b calculates a modulo b (i. e. the remainder after a/b). See also Fig. 3.

```
inc := iSampNo mod N;
for i := 0 step 1 until N-1 do
    begin
    shift[(N*i+inc) mod N] := 0;
    for k := -NGroups step 1 until NGroups-1 do
       shift[(N*i+inc) mod N] := shift[(N*i+inc) mod N] +
          X[isampNo + k*N + i] * Imp[k*N + i];
    end;
```

Fig. 3. In most of the code listings in this article, we will not be concerned with paring down the code to its tightest possible state. For example, we have not bothered to define a variable **pi2** to hold the value 2π, which is needed in several different places. However, we should point out that both the filtering, time-aliasing and rotation of Fig. 2 can be accomplished inside one loop, as is shown here.







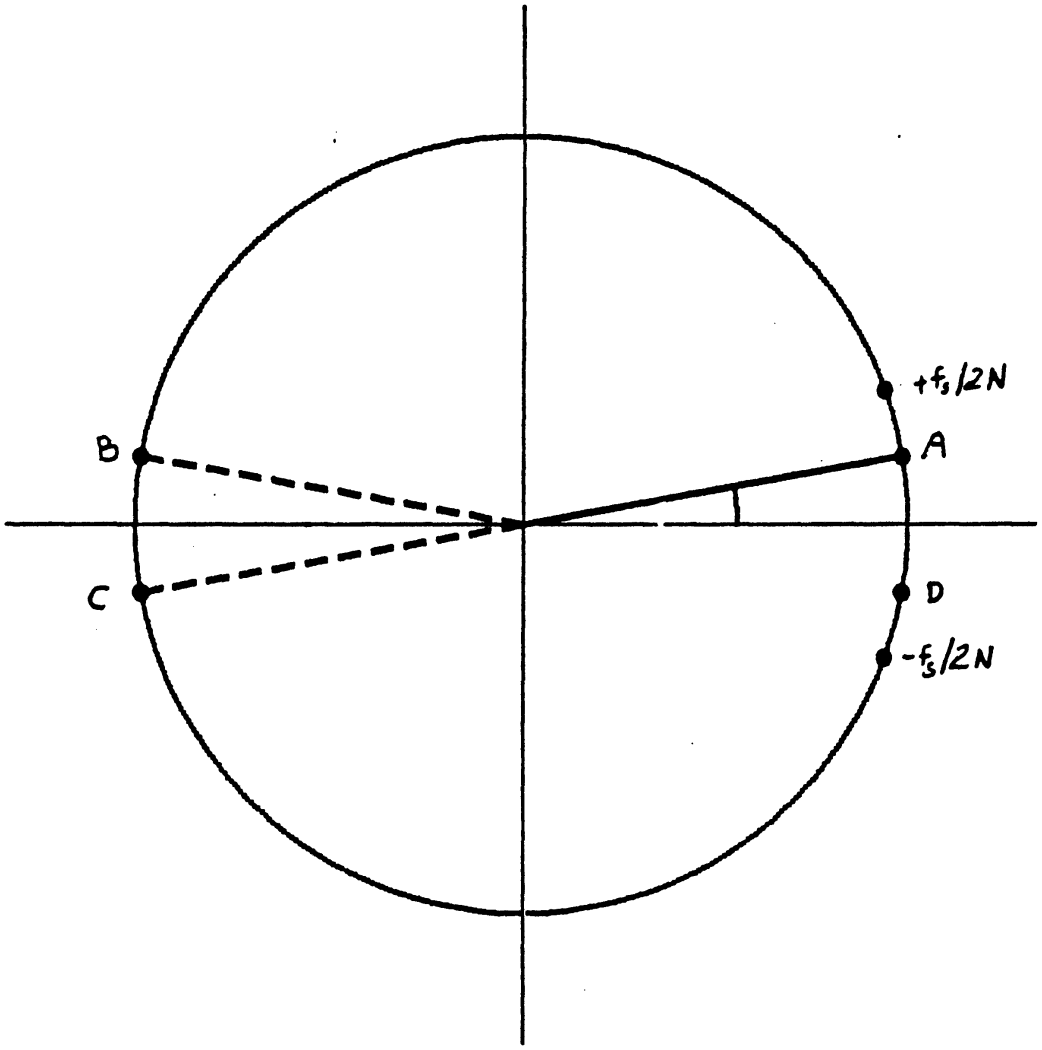
```
procedure convert(real array realPart,imagPart; integer nAnalPts,N,R,Q,nQs;
                  real array magInterm,freqInterm);
   begin "convert"
   comment
   INPUTS
   realPart,imagPart dimensioned [0:N/2,0:nAnalPts-1], containing real and
                     imaginary parts of analysis output at sRate/R
   nAnalPts          the largest integer in (nSamps-1)/R + 1
   N,R               see text
   Q                 interpolation factor for intermediate sampling rate sRate/Q
                     Q must divide evenly into R!
   nQs               number of groups of Q samples in 1/2 of intLen (defined below)
   OUTPUTS:
   magInterm, freqInterm   dimensioned [0:N/2, 0:nIntermPts], containing magnitude
                           and frequency values (calculated as angle difference
                           values) for each analysis channel at intermediate
                           sampling rate sRate/Q. nIntermPts is defined below.
   ;
   integer
      ROverQ,    comment R/Q, used to interpolate to intermediate sampling rate;
      intLen,    comment length of impulse response of interpolating filter;
      winLen,    comment length of positive half of window, including the sample
                         in the middle of the window;
      nIntermPts, comment no. of points in current channel at intermediate sampling rate;
      offset, inc,
                 comment indices for interpolation to intermediate sampling rate;
      chanNo,    comment index for channel number;
      sampNo,    comment index for sample number;
      m;         comment index for convolution;
   boolean
      magInvert,      comment For tracking 180-degree phase shifts;
      saved;          comment For skipping over consecutive (0,0) points;
   real
      thisReal,thisImag,
      thisTheta,           comment θ(n);
      lastTheta,           comment θ(n - 1);
      saveTheta,           comment For skipping over consecutive (0,0) points;
      thetaDiff,           comment dθ/dt;
      tempMag,             comment magnitude at n --- might be inverted;
      lastMag,             comment magnitude at n-1 --- might be inverted;
      intMult,             comment multiplier used in interpolating to intermediate
                                   sampling rate;
      beta;                comment parameter for Kaiser window;
```

```
real array tempReal,tempImag[-nQs : nAnalPts+nQs-1];
          comment real  and Imag for current channel at sampling rate sRate/R,
          padded on both ends with nQs zeros;
real array win[0 : Q*nQs];
real array interp[-Q*nQs : Q*nQs];
          comment win holds positive half of window and interp holds
          impulse response for interpolating filter;
real array intermReal,intermImag[0:(nAnalPts-1)*Q];
          comment intermReal and intermImag hold realPart and imagPart,
          respectively, interpolated to intermediate sampling rate sRate*Q/R;

beta := 6.8;          comment about 71 dB down;
winLen := Q*nQs+1;
intLen := 2*Q*nQs + 1;
ROverQ := R/Q;
nIntermPts := (nAnalPts-1)*Q+1;

comment stuff array win with half of the symmetric Kaiser window;
Kaiser(intLen,win,winLen,1,beta);    comment "1" means that impLen is odd;

comment windoW impulse response of interpolating lowpass filter;
interp[0] := win[0];
for sampNo := 1 step 1 until Q*nQs do
   interp[sampNo] := interp[-sampNo] := Q*win[sampNo]*SIN(pi*sampNo/Q)/(pi*sampNo);

for chanNo := 0 step 1 until N/2 do
   begin "channel loop"

   comment fill tempReal and tempImag with realPart and imagPart surrounded
      by zeros;
   for sampNo := -nQs step 1 until -1 do
      begin
      tempReal[sampNo] := 0;
      tempImag[sampNo] := 0;
      end;
   for sampNo := 0 step 1 until nAnalPts-1 do
      begin
      tempReal[sampNo] := realPart[chanNo,sampNo];
      tempImag[sampNo] := imagPart[chanNo,sampNo];
      end;
   for sampNo := nAnalPts step 1 until nAnalPts+nQs-1 do
      begin
      tempReal[sampNo] := 0;
      tempImag[sampNo] := 0;
      end;

   for sampNo := 0 step 1 until nIntermPts-1 do
      begin "interpolate to intermediate sampling rate"
```

```
      offset := sampNo mod Q;
      inc := (sampNo div Q);   comment inc := largest integer in sampNo/Q;
      intermReal[sampNo] := 0;
      intermImag[sampNo] := 0;

      for m := -nQs+1 step 1 until nQs do
         begin      comment perform the interpolation as a convolution;
         intMult := interp[m*Q - offset];
         intermReal[sampNo] := intermReal[sampNo] + tempReal[m + inc] * intMult;
         intermImag[sampNo] := intermImag[sampNo] + tempImag[m + inc] * intMult;
         end;

      end "interpolate to intermediate sampling rate";

   comment Convert to magnitude and angle difference (frequency) at
           intermediate sampling rate;
   lastMag := 1.0;
   lastTheta := 0.0;
   saved := magInvert := false;

   for sampNo := 0 step 1 until  nIntermPts-1 do
      begin
      thisReal := intermReal[sampNo];
      thisImag := intermImag[sampNo];
      tempMag := (thisReal*thisReal + thisImag*thisImag)↑0.5;
```

```
        if tempMag = 0 then
           begin      comment Don't let angle change;
           if not saved then
              begin "Savem"
              saveTheta := lastTheta;
              saved := true;
              end "Savem";
           FreqInterm[chanNo,sampNo] := 0;
           end
        else begin "Get Angle"
           if lastMag = 0 then
              begin      comment Ignore the previous zero point in calculating
                                 current angle;
              lastTheta ← saveTheta;
              saved ← false;
              end;
            thisTheta ← ATAN2(thisImag,thisReal);
            thetaDiff ← thisTheta - lastTheta;
            while thetaDiff > pi/2 do
               begin
               thetaDiff ← thetaDiff - pi;
               magInvert ← not magInvert;
               end;
            while thetaDiff < -pi/2 do
               begin
               thetaDiff ← thetaDiff + pi;
               magInvert ← not magInvert;
               end;
            FreqInterm[chanNo,sampNo] ← thetaDiff/ROverQ;
            end "Get Angle";
        magInterm[chanNo,sampNo] ← (if magInvert then -tempMag else tempMag);
        lastMag ← tempMag;
        lastTheta ← thisTheta;
        end;
     end "channel loop";
  end "convert";
```

Fig. 7. Conversion of data from real-imaginary form to amplitude-frequency form. The operator *a* **div** *b* calculates the number of times *b* divides into *a*. **ATAN2(a,b)** takes the arc tangent of *a*/*b*, preserving sign (and thus quadrant) information (see text). Note that the arc tangent function supplied with many compilers does not preserve this sign information.


+fs/2N
A
B
C
D
-fs/2N

```
procedure toHz(real array freqInterm; integer nIntermPts,N,R,sRate;
  real array outFreq);
begin
   comment
   INPUTS
   freqInterm  dimensioned [0:N/2, 0:nIntermPts-1],
               containing  frequency values for each analysis channel at
               intermediate sampling rate in radians per sec.
   nIntermPts  length of freqInterm and outFreq at intermediate sampling
               rate sRate*Q/R.
   N, R        see text
   sRate       original sampling rate
   OUTPUTS
   outFreq     storage/examination array dimensioned [0:N/2, 0:nIntermPts-1]
               containing frequency values in Hz.
   ;
   integer chanNo,      comment index for channel number;
      sampNo;           comment index for sample number;
   real centerFreq,     comment center frequency of channel in Hz;
      radToHz;          comment convert radians per sec to cycles per sec (Hz);
   radToHz := sRate / (2*pi);
   for chanNo := 0 step 1 until N/2 do
      begin "channel loop"
      centerFreq := sRate*chanNo/N;
      for sampNo := 0 step 1 until nIntermPts-1 do
         outFreq[chanNo,sampNo] := freqInterm[chanNo,sampNo] * radToHz + centerFreq;
      end "channel loop";
   end;
```

Fig. 9. Changing the format of the frequency information from radians per second to Hertz for storage or examination.

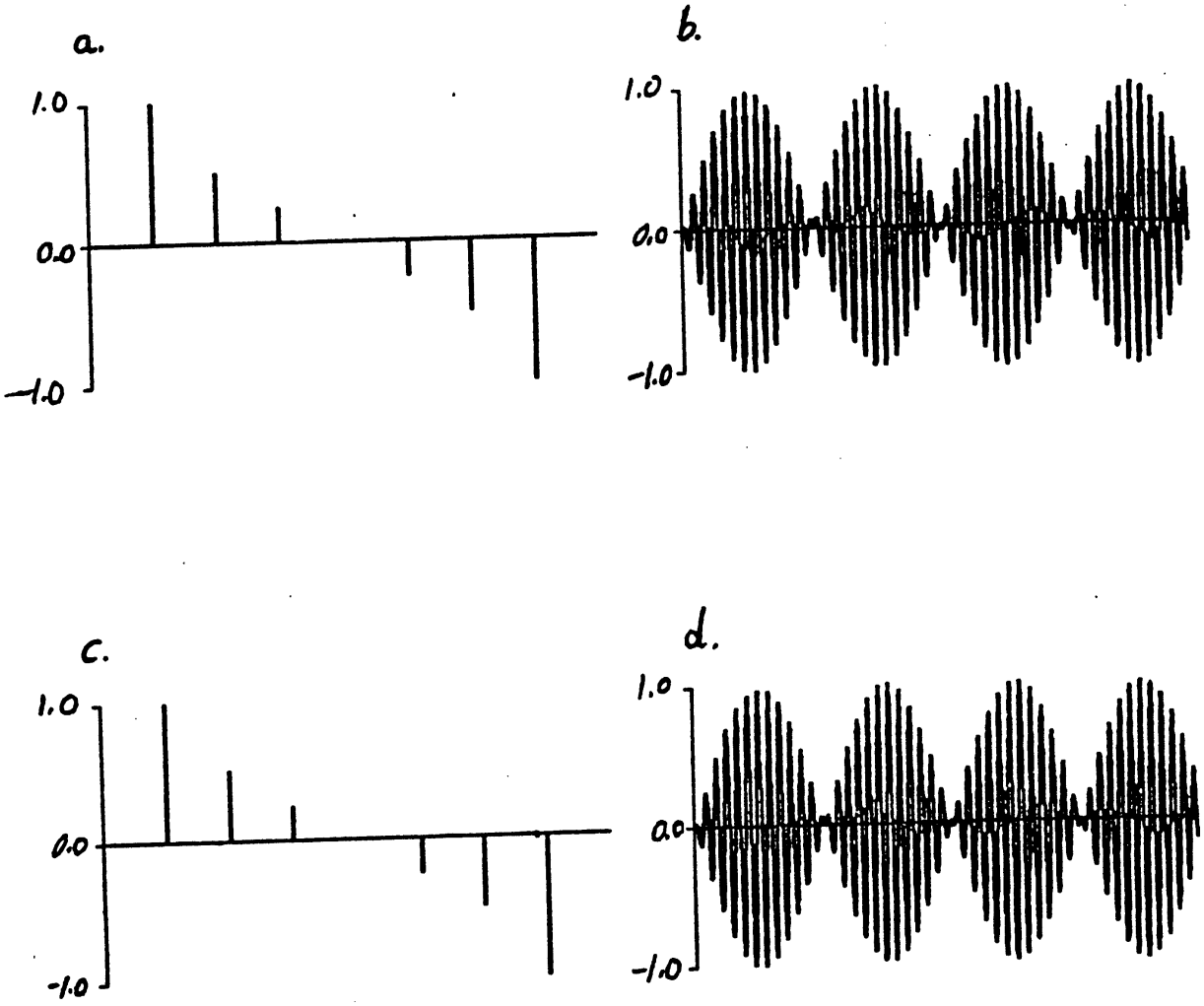
```
procedure fromHz(real array outFreq; integer nIntermPts,N,R,sRate;
                 real array freqInterm);
begin
   comment
   INPUTS
   outFreq     storage/examination array dimensioned [0:N/2, 0:nIntermPts-1]
               containing frequency values in Hz.
   nIntermPts  length of freqInterm and outFreq at intermediate sampling
               rate sRate*Q/R.
   N, R        see text
   sRate       original sampling rate
   OUTPUTS
   freqInterm  dimensioned [0:N/2, 0:nIntermPts-1],
               containing  frequency values for each analysis channel at
               intermediate sampling rate in radians per sec.
   ;
   integer chanNo,      comment index for channel number;
      sampNo;           comment index for sample number;
   real centerFreq,     comment center frequency of channel in Hz;
      hzToRad;          comment convert Hz to radians per sec;
   hzToRad := 2*pi / sRate;
   for chanNo := 0 step 1 until N/2 do
        begin "channel loop"
        centerFreq := sRate*chanNo/N;
        for sampNo := 0 step 1 until nIntermPts-1  do
           freqInterm[chanNo , sampNo] :=
              (outFreq[chanNo , sampNo] - centerFreq)*hzToRad;
        end "channel loop";
   end;
```

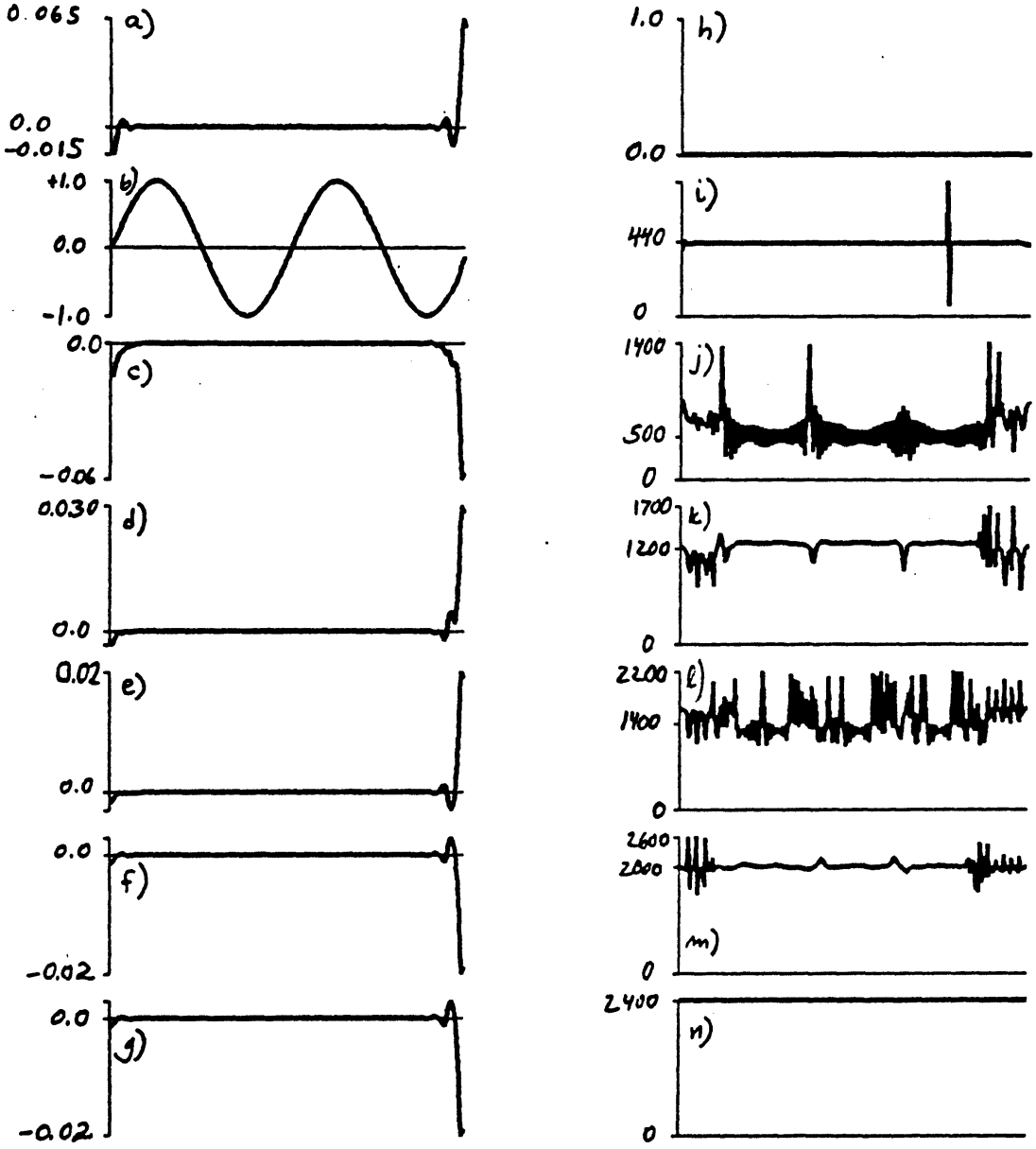
Fig. 10. Changing the format of the frequency information from Hertz back to radians per second for synthesis.

```
procedure synthesize(real array magInterm,freqInterm; integer nIntermPts,N,R,Q;
   real freqMult; real array sound);
begin
   comment
   INPUTS
   magInterm, freqInterm   dimensioned [0:N/2, 0:nIntermPts-1]
                           containing magnitude and frequency values for each
                           analysis channel at intermediate sampling rate
   nIntermPts              at intermediate sampling rate sRate*Q/R
   N, R                    see text
   Q                       compression ratio for intermediate sampling rate
   freqMult                frequency multiplier --- see text
   OUTPUTS:
   sound                   output sound file at original sampling rate,
                           dimensioned [0:nOSamps-1], with nOSamps defined below
   ;
   integer
      nOSamps,          comment number of samples in resynthesized output;
      chanNo,           comment index for channel number;
      sampNo,           comment index for sample number;
      m;                comment index for linear interpolation to original sRate;
   real
      centerFreq,       comment for determining absolute frequency for each
                                channel, in radians/sample;
      oldPhase, phase,  comment for determining instantaneous phase angle;
      scale,            comment to accomodate for chanNo = 0 and N/2;
      ROverQ,           comment R/Q, used for linear interpolation  to sRate;
      freqSlope, magSlope;      comment for linear interpolation to sRate;
   real array mag,freq[0 : (nIntermPts-1)*R/Q];
                              comment To hold a channel's mag and freq
                                     arrays interpolated to sRate;
   ROverQ := R/Q;
   nOSamps := (nIntermPts-1)*R/Q + 1;
   comment Make sure that array sound is empty;
   for sampNo := 0 step 1 until nOSamps-1 do
      sound[sampNo] := 0;
   for chanNo := 0 step 1 until N/2 do
      begin "channel loop"
```

```
    for sampNo := 1 step 1 until nIntermPts-1 do
       begin comment interpolate linearly to original sampling rate;
       freqSlope := (freqInterm[chanNo,sampNo] - freqInterm[chanNo,sampNo-1])/ROverQ;
       magSlope  := (magInterm[chanNo,sampNo]  - magInterm[chanNo,sampNo-1]) /ROverQ;
       for m := 0 step 1 until ROverQ-1 do
          begin "interpolation loop"
          mag[(sampNo-1)*ROverQ + m] := magInterm[chanNo,sampNo-1] + magSlope*m;
          freq[(sampNo-1)*ROverQ + m] := freqInterm[chanNo,sampNo-1] + freqSlope*m;
          end;
       end;
    comment Set last points;
    mag[(nIntermPts-1)*ROverQ] := magInterm[chanNo,nIntermPts-1];
    freq[(nIntermPts-1)*ROverQ] := freqInterm[chanNo,nIntermPts-1];

    comment initialization for current channel;
    centerFreq := 2*pi*chanNo/N;
    scale := (if (chanNo = 0 or chanNo = N/2) then 0.5 else 1.0);
       comment All channels represent both positive and negative frequencies
               except channels numbered 0 and N/2;
    oldPhase := -centerFreq * FreqMult;

    for sampNo := 0 step 1 until nOSamps-1 do
       begin "additive synthesis"
       phase := oldPhase + freqMult*(freq[sampNo] + centerFreq);
       sound[sampNo] := sound[sampNo] + scale * mag[sampNo] * COS(phase);
       oldPhase := phase;
       end "additive synthesis";

    end "channel loop";

end;
```

Fig. 11. Resynthesis from the analysis data, with possible transposition of original frequencies.


a.
1.0
0.0
-1.0
b.
1.0
0.0
-1.0
c.
1.0
0.0
-1.0
d.
1.0
0.0
-1.0


0.065
a)
0.0
-0.015
+1.0
b)
0.0
-1.0
0.0
c)
-0.06
0.030
d)
0.0
0.02
e)
0.0
0.0
f)
-0.02
0.0
g)
-0.02
1.0
h)
0.0
i)
440
0
1400
j)
500
0
1700
k)
1200
0
2200
l)
1400
0
2600
2000
m)
0
2400
n)
0