14 Digital Audio Compression

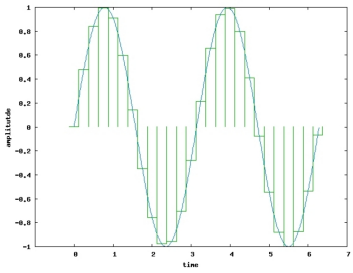
The compression of digital audio data is an important topic. Compressing (reducing) the data storage requirements of digital audio allows us to fit more songs into our iPods and download them faster. We will apply ideas from interpolation, least-squares approximation, and other topics, in order to reduce the storage requirements of digital audio files. All of our approaches replace the original audio signal by approximations that are made up by a linear combination of cosine functions.

We describe basic ideas behind data compression methods used in mp3 players. The numerical experiments use several interesting MATLAB/Octave commands, including commands for the manipulating sound.

14.1 Computers and sound

Sound is a complicated phenomenon. It is normally caused by a moving object in air (or other medium), for example a loudspeaker cone moving back and forth. The motion in turn causes air pressure variations that travel through the air like waves in a pond. Our eardrums convert the pressure variations into the phenomenon that our brain processes as sound.

Computers "hear" sounds using a microphone instead of an eardrum. The microphone converts pressure variations into an electric potential with amplitude corresponding to the intensity of the pressure. The computer then processes the electrical signal using a technique called sampling. Computers sample the signal by measuring its amplitude at regular intervals, often 44,100 times per second. Each measurement is stored as a number with fixed precision, often 16 bits. The following diagram illustrates the sampling process showing a simple wave sampled at regular intervals[1]:



Sample values
0.000000000000000
0.479425538604203
0.841470984807897
0.997494986604054
0.909297426825682
0.598472144103956
0.141120008059867
-0.35078322768962
-0.756802495307928
-0.977530117665097
-0.958924274663138
...

Computers emit sound by more or less reversing the above process. Samples are fed to a device that generates an electric potential proportional to the sample values. A speaker or other similar device may then convert the electric signal into air pressure variations.

The rate at which the measurements are made is called the *sampling rate*. A common sampling rate is 44,100 times per second (used by compact disc, or CD, audio). The format of numbers used to store the sampled audio signal generally differs from the floating point numbers described in Lecture 2, with 64 bits per number. For example, compact discs use 16 bit numbers to store the samples.

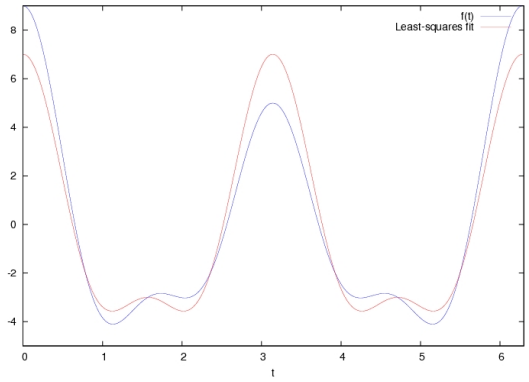
The *bit rate* of a set of digital audio data is the storage in bits required for each second of sound. If the data has fixed sampling rate and precision (as does CD audio), the bit rate is simply their product. For example, the bit rate of one channel of CD audio is 44,100 samples/second $\times$ 16 bits/sample = 705,600 bits/second. The bit rate is a general measure of storage, and is not always simply the product of sampling rate and precision. For example, we will discuss a way of encoding data with variable precision.

[1] Image adapted from Franz Ferdinand, ©2005

Large storage requirements limit the amount of audio data that can be stored on compact discs, flash memory, and other media. Large file sizes also give rise to long download times for retrieving songs from the internet. For these reasons (and others), there is considerable interest in shrinking the storage requirements of sampled sound.

14.2 Least-squares data compression

Least-squares data fitting can be thought of as a method for replacing a (large) set of data with a model and a (smaller) set of model coefficients that approximate the data by minimizing the norm of the difference between the data and the model.



Consider the following simple example. Let the function $f(t) = \cos(t) + 5\cos(2t) + \cos(3t) + 2\cos(4t)$. A plot of $f(t)$ for $0 \leq t \leq 2\pi$ appears as the blue curve in the figure. Assume that we are given a data set of 1000 discrete function values of $f(t)$ regularly spaced over the interval $0 \leq t \leq 2\pi$. We can fully interpolate the data by setting up the model matrix A (using MATLAB/Octave notation):

```
t = linspace (0,2*pi,1000)';
b = cos(t) + 5*cos(2*t) + cos(3*t) + 2*cos(4*t);
A = [ones(size(t)), cos(t), cos(2*t), cos(3*t), cos(4*t)];
```

and then solving the linear system $Ax = b$ with the command `x=A\b`. Try it! Note that the solution vector components match the function coefficients.

Some of the coefficients are not as large as others in this simple example. We can approximate the function f with a least-squares approximation that omits parts of the model corresponding to smaller coefficients. For example, set up the least-squares model

```
A = [cos(2*t), cos(4*t)];
```

and solve the corresponding least-squares system `x=A\b`. This model uses only two coefficients to describe the data set of 1000 data points. The resulting fit is reasonable, and is displayed by the red curve in the figure. The plot was made with the command `plot(t,b,'-b',t,A*x,'-r')`.

The cosine function oscillates with a regular frequency. The multiples of t in the above example correspond to different frequencies (the larger the multiple of t is, the higher the frequency of oscillation). The least-squares fit computed the best approximation to the data using only two frequencies.

Exercise 14.1

Experiment with different least-squares models for the above example by omitting different frequencies. Plot your experiments and briefly describe the results. □

14.3 Manipulating sound in MATLAB and Octave

MATLAB and Octave provide several commands that make it relatively easy to read in, manipulate, and listen to digital audio signals. This lecture is accompanied by a short sound file

`http://www.math.kent.edu/ blewis/numerical_computing.1/project5/shostakovich.wav`.

The data of the file is sampled at 22,050 samples per second and 16 bits per sample (1/2 the bit rate of CD audio), and is corresponds to 40 seconds of music. If you do not care for the tune, you are free to experiment with any audio samples that you wish. In order to experiment with the provided file, you will need to download it from the above link into your working directory. The MATLAB/Octave command to load an audio file is:

```
[b,R] = wavread ('shostakovich.wav');
N = length(b);
```

The returned vector b contains the sound samples (it's very long!), R is the sampling rate, and N is the number of samples. Note that even though the precision of the data is 16 bits, MATLAB and Octave represent the samples as double-precision internally. You can listen to the sample you just loaded with the command:

```
sound (b,R);
```

Some versions of MATLAB and Octave may have slightly different syntax; use the help command for more detailed information. The `wavread` command for Octave is part of the Octave-Forge project.

Sampled audio data is generally much more complicated looking than the simple example in the last section; view for instance the data of the read file. This can be done with the command `plot(b)`. However, also the data of the file can be interpolated or fitted in the least-squares sense with a cosine model

$$y = c_0 + c_1 \cos(\omega_1 t) + c_2 \cos(\omega_2 t) + \cdots + c_{n-1} \cos(\omega_{n-1} t),$$

for some positive integer $n-1$ and frequencies ω_j. An important result from information theory, the Shannon-Nyquist theorem, requires that the highest frequency in our model, ω_{n-1}, be less than half the sampling rate. That is, our cosine model assumes that the audio data is filtered to cut-off all frequencies above half the sampling rate.

The cosine model requires additional technical assumptions on the data. Recall that the cosine function is an even function, and the sum of even functions is an even function. Therefore, the model also assumes that the data is even. The usual approach taken to satisfy this requirement of the model is to simply assume that the data is extended outside of the interval of interest to make it even.

The above-mentioned conditions (cut-off frequency, extension beyond the interval boundaries) are in general important to consider, but we will not discuss the details in this lecture. Instead, we focus on the basic ideas behind compression methods, such as mp3.

14.4 Computing the model interpolation coefficients with the DCT

Let the vector b contain one second of sampled audio, and assume that the sampling rate is N samples per second (b is of length N). It is tempting to proceed just as in the simple example above by setting up an interpolation model

```
t = linspace (0,2*pi,N)';
A = [ones(size(t)), cos(t), cos(2*t), cos(3*t), ..., cos((N/2-1)*t)];
x = A\b;
```

Aside from a few technical details, this method could be used to interpolate an audio signal. However, consider the size of the quantities involved. At the CD-quality sampling rate, $N = 44100$, and the matrix A is gigantic (44100×22050)! This problem is unreasonably large.

Fortunately, there exists a remarkable algorithm, the Fast Discrete Cosine Transform (DCT), which can compute the solution efficiently. The DCT is a variation on the FFT method described in Lecture 13. The DCT produces scaled versions of the model coefficients with the command:

```
c = dct(b);
```

The computed coefficient vector c has the same number of components as b.

To investigate the plausibility of the DCT, we can try it out on our simple example:

```
% Simple example revisited
t = linspace (0,2*pi,1000)';
b = cos(t) + 5*cos(2*t) + cos(3*t) + 2*cos(4*t);
x = dct(b);
N = length(b);
w = sqrt(2/N);
f = linspace(0, N/2, N)';
plot (f(1:8),w*x(1:8),'x');
```

The variable w is a scaling factor produced by the DCT algorithm and the vector f is the frequency scale for the model coefficients computed by the DCT and stored in x. The frequency range from 0 to $N/2 - 1$ corresponds to half the sampling rate (assumed here to be N). We can think of the `dct(b)` command as essentially computing `A\b` for the full interpolation model using the frequencies in the vector f. Your plot should show that we closely compute the model coefficients (i.e., a value of 1 at frequency 1, 5 at frequency 2, etc.)

We can reconstruct the original signal from the model coefficients with the command:

```
y = idct(x);     % The reconstructed data is in y.
plot (t, b, '-r', t, y,'-b');
```

The plots should overlay each other. The `idct` command is the inverse of the `dct` command. We can think of `idct(x)` as computing the product Ax for an appropriate model matrix A and coefficient vector x.

14.5 Digital filtering

The DCT algorithm can be used to not only interpolate data, but to compute a least-squares fit to the data by omitting frequencies. The process of computing a least-squares fit to digitized signals by omitting frequencies is called digital filtering. Digital filtering can reduce the storage requirements of digital audio by simply lopping off parts of the data that correspond to specific frequencies. Of course, cutting out frequencies affects the sound quality of data. However, the human ear is not equally sensitive to all frequencies. In particular, we generally do not perceive very high and very low frequencies nearly as well as mid-range frequencies. In some cases, we can filter out these frequencies without significantly affecting the perceived quality. An easy way to filter specific frequencies in MATLAB and Octave is to generate a mask. Consider the example:

```
[b,R] = wavread ('shostakovich.wav');
```

```
N = length(b);
c = dct(b);                % Compute the interpolation model coefficients
w = sqrt(2/N);
f = linspace(0,R/2,N)';
plot (f,w*c);     % Shows a plot of the frequencies coefficients for the sample
% Generate a mask of zeros and ones. m is 0 for every frequency above 3000, 1 otherwise.
% This mask will cut-off all frequencies above 3000 cycles/second.
m = (f<3000);
plot (f,w*m.*c);   % Display the filtered frequency coefficients.
y = idct(m.*c);    % Generate a filtered sound sample data set
sound(y,R);        % Listen to the result
```

Exercise 14.2

Experiment with several frequency cut-off values in the above example. Listen to your results. □

Exercise 14.3

Exhibit how to construct a single mask that will cut off frequencies below 200 and above 5000 cycles/second. □

Exercise 14.4

How much does the above code reduce the storage requirement of the sample (in bit rate)? □

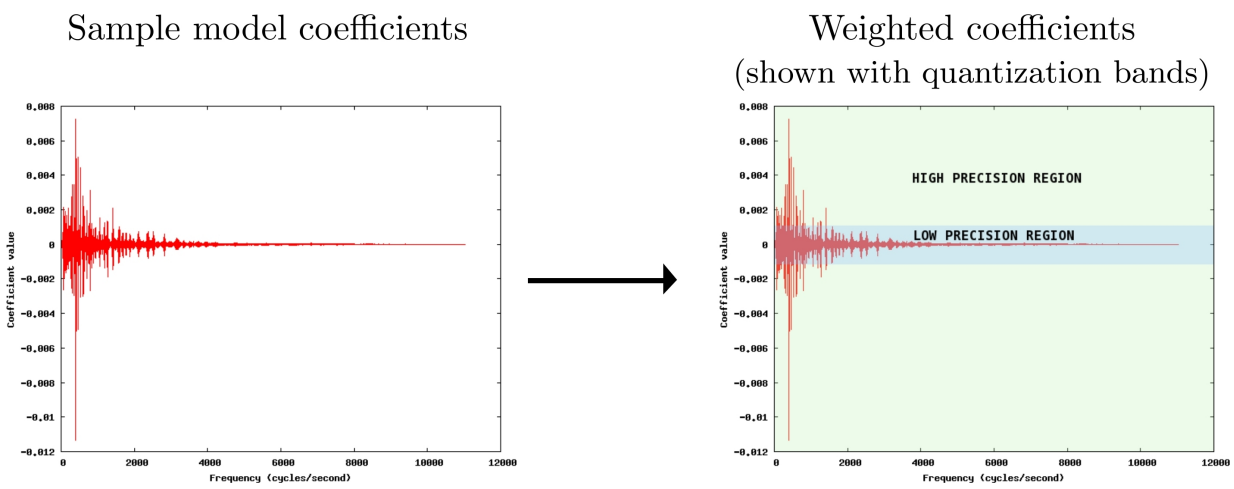
14.6 The ideas behind mp3

Digital filtering is an effective technique for compressing audio data in many situations, especially telephony. Cutting out entire frequency ranges is rather a brute-force method, however. There are more effective ways to reduce the storage required of digital audio data, while also maintaining a high-quality sound.

One idea is this: rather than cutting out "less-important" frequencies altogether, we could store the corresponding model coefficients with lower precision - that is, with fewer bits. This technique is called quantization. The "less-important" frequencies are determined by the magnitude of their DCT model coefficients. Coefficients of small magnitude correspond to cosine frequencies that do not contribute much to the sound sample. A key idea of methods like the mp3 algorithm is to focus the compression on parts of the signal that are perceptually not very important.

Here is an illustration of an audio compression method similar to (but much simpler than) mp3 compression: Note that with the illustrated choice of quantization bands, the bulk of the model coefficients lie in the low precision storage realm in the above example. Our compression method will encode all of the corresponding unweighted model coefficients that fall in that band with low precision. The precise cut-off between low- and high-precision storage will govern the overall compression obtained by this method.

For example, assume that 90% of the coefficients lie in the low-precision part of the illustration. Suppose that we store those coefficients with only 8-bit numbers, and the remaining ones with 16-bit numbers. The resulting data will only require about 55% of the storage space used by the original sample data set of entirely 16-bit numbers.



Exercise 14.5

What is the bit rate of the compressed audio sample discussed in the last paragraph, assuming 22,050 samples per second? □

We can achieve higher compression by either widening the low-precision region, or by lowering the precision used to store the coefficients, or both. The algorithm used in mp3 compression uses similar techniques to achieve up to a 10:1 compression of CD audio and still maintain a high perceived quality of sound.

14.7 Quantization in MATLAB and Octave

MATLAB and Octave do not easily represent quantized numbers internally. We can, however, simulate the result of quantization in double-precision with the function `quantize.m`:

```
function y = quantize (x, bits)

m = max(abs(x));
y = x/m;
y = floor((2^bits - 1)*y/2);
y = 2*y/(2^bits -1);
y = m*y;
```

Exercise 14.6

Explain how the function quantize.m works. □

14.8 MP3-like compression with MATLAB and Octave

The following code illustrates our discussion of audio data compression with an actual audio. The code requires the function `quantize.m`.

```
% Load an audio sample data set
[b,R] = wavread (shostakovich.wav);
N  = length(b);
% Compute the interpolation model coefficients
c = dct(b);
w = sqrt(2/N);
f = linspace(0,R/2,N)';
% Lets look at the weighted coefficients and pick a cut-off value
plot (f,w*c)
% Pick a cut-off value and split the coefficients into low- and high-precision sets:
cutoff = 0.00015
mask = (abs(w*c)<cutoff);
low=mask.*c;
high=(1-mask).*c;
% This plot nicely illustrates the cut-off region:
plot(f,w*high,'-R',f,w*low,'-b')
% Now pick a precision (in bits) for the low precision data set:
lowbits=8
% We wont quantize the high-precision set of coefficients (high), only the
% low precision part (requires quantize.m):
low = quantize(low, lowbits);
% Finally, let's reconstruct our compressed audio sample and listen to it!
y=idct(low+high);
sound (y,R);
```

Exercise 14.7

Experiment with the above code, trying out different cut-off values and precision values (lowbits). Listen to your results. What is the lowest bit rate that you can find that still sounds good to you? □