

Sampling Rate Conversion

Overview

In discrete-time signal processing, it is often necessary to convert a signal from one sampling rate to another. A common example is the conversion from the sampling rate of a compact disk (CD); signal (44.1 KHz) to that of a Digital Audio Tape (DAT) signal (48 KHz). Another example is the audio standard in High-Definition Television (HDTV) transmission, where at least three sampling rates are supported (32, 44.1, and 48 KHz). Although in principle we may convert the signal back to analog form and resample at the desired rate, it is usually preferable to perform the entire conversion digitally. This is due to many considerations including the fact that conversion to analog form often introduces noise in the signal, and that digital signal processing can be much more cost-effective and flexible.

This MATLAB project asks you to perform a sampling rate conversion on segments of audio signals. The input audio signals are quantized to 8 bits and sampled with a sampling frequency of 11,025 Hz. You are required to convert the signal to a sampling rate of 24,000 Hz in a computationally efficient manner. Although one conceptual way of realizing this sampling rate conversion process is to upsample the signal, lowpass filter, and downsample it, a cleverer implementation can lead to an implementation that is many times more efficient. To do this, you can exploit various aspects of class to optimize the system, such as multistage filter implementation, filter design, and polyphase implementation. By the end of the project, you hopefully will have a much better understanding of both the theoretical aspect of the system as well as various issues in implementing a practical DSP system at a software level.

Project Goal

A sampling rate converter which produces an output signal with a sampling rate which is $\frac{L}{M}$ times the original sampling rate can be specified as shown in Figure 1.

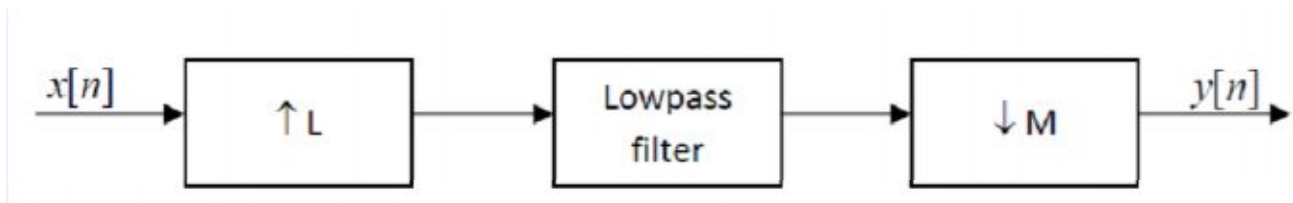


Figure 1: Sampling rate conversion system.

For an ideal sampling rate converter, the lowpass filter in Figure 1 is an ideal lowpass filter with cutoff frequency $\omega_c = \min(\frac{\pi}{M}, \frac{\pi}{L})$. The goal of this project is to design an efficient DSP algorithm that implements the system in Figure 1 subject to the following constraints:

The system performs the correct sampling rate conversion from 11,025 Hz to 24,000 Hz. In particular, do not assume that $11,025 \text{ Hz} \approx 11,000 \text{ Hz}$.

If the system in Figure 1 is an ideal sampling rate converter, when the input $x[n]$ is a unit impulse $\delta[n]$, the output $y[n]$ has a Fourier transform $Y(e^{j\omega})$ which corresponds to an ideal lowpass filter with a cutoff frequency at $\omega_c = (11025/24000)\pi$. For an equivalent system which you are to implement, when the input $x[n]$ is a unit impulse $\delta[n]$, the output $y[n]$, must have a Fourier transform $Y(e^{j\omega})$ which is an approximation of a lowpass filter, and meets the specifications shown below in Table 1.

Passband Cutoff (ω_p)	$\frac{11,025\pi}{24,000}$
Passband Ripple	$\pm 0.1 \text{ dB}$ or less
Stopband Frequency (ω_s)	$1.2\omega_p$
Stopband Attenuation	70 dB or more
Phase Constraints	$ \max \text{grpdelay} - \min \text{grpdelay} \leq 720$ in the passband

Table 1

If your sampling rate conversion system functions properly, you should meet the specifications in Table 1, and when you play the output audio signal at 24,000 Hz, it should sound the same as the input audio signal played at 11,025 Hz.

To digitally verify that your system does indeed meet the specifications, you must pass $y[n]$, the response of the system when $x[n] = \delta[n]$, to the function `verify.m`. This MATLAB function will make measurements on your output and verify that it meets all the specifications given above. In your report you must attach a hard copy of the plot generated when calling `verify.m`. Details on project write-up will be discussed in later sections.

You should get a rough estimate of the efficiency of your design by determining how much computation was required to perform the sampling rate conversion on the **Wagner.wav** signal. The method which you will use to count the number of operations will be described shortly.

You are to write a MATLAB function `srconvert` such that the command `srconvert(in)` takes the input signal `in` with an associated sampling rate of 11,025Hz, and returns an output signal at a sampling rate of 24,000Hz. (See the section on writing MATLAB functions.) Once your function `srconvert.m` is finalized, run the command `y=srconvert([1 zeros(1,3000)])`. This will produce a vector `y` which contains the response of your system to a unit impulse. Then call `verify(y)` to verify that your design meets the design specification.

The Files

The project zip file contains the audio files and MATLAB functions which you will need for this project.

In testing your system, you may find it helpful to first use a unit impulse as input to your system and then later try using real audio signals as inputs and listening to the outputs. To load a sound file into a vector **x**, type `x=wavread('filename.wav');`. To write a sound file for the MATLAB vector **x** into the current directory, type `wavwrite(x, sampfreq, 'filename.wav')`. You may test your system using any of the audio signals, although you should use the Wagner.wav signal to benchmark your system performance.

MATLAB Utilities

You may find the following MATLAB functions useful for your design.

`examplpf(h, wp, ws)` - allows you to examine the passband ripple, the group delay, as well as the stopband attenuation of the lowpass filter whose impulse response is **h**. It generates three simultaneous plots. The first one zooms in on the passband with cutoff frequency **wp**. The second plot measures the group delay in the passband, and the third plot is simply the magnitude response over the entire frequency range $(-\pi : \pi)$. The passband (**wp**) and stopband (**ws**) cutoff frequencies are normalized by π . That is, a cutoff frequency at $\pi/2$ should be entered as 0.5.

`poly1(h, M)` - returns a matrix **E** whose i^{th} row corresponds to the i^{th} polyphase components of the FIR filter **h**, obtained via type **I** decomposition with downsampling factor **M**.

`fftfilt(h, x)` - convolves the signal **x** with the filter **h** using the FFTs. Beware that this command might not always leads to a more efficient implementation than `conv(h, x)`, depending on the length of the signals.

`upsample(h, L)` - returns an upsampled version of **h** by a factor **L**, without any interpolation.

`downsample(h, M)` - downsamples **h** by **M**.