University of Manchester
School of Computer Science

COMP28512: Mobile Systems

Semester 2 – 2018-19

Laboratory Task 2

Frequency-domain processing

10 Feb 2019

Introduction

Frequency-domain processing is used in smart-phones for music and image digitisation, and also for generating and receiving the required radio transmissions. Transform coding techniques are used to digitise wide-band speech and music with reasonable economy in the bit-rate or storage capacity required. Transmitting or storing CD quality music on mobile devices is probably still too expensive in terms of bit-rate, hence the need for bit-rate reduction. Frequency-domain techniques are used in smart-phones for mp3 audio and JPEG image coding. They are also used by DAB radios, digital television and internet radio streaming.

This task starts with a look at frequency-domain concepts, starting with the Fourier series and continuing to frequency-domain processing and the principles of psycho-acoustic coding as used by mp3. After the Week 4 laboratory session, you will be asked to submit your work and your code to Blackboard. You will also be required to give a brief demonstration and answer questions on what you have achieved.

Some background

Waveforms may be converted into the frequency-domain using forms of the Discrete Fourier transform (DFT). A highly efficient implementation of the DFT is known as the Fast Fourier transform (FFT). The FFT has revolutionised many aspects of digital signal processing (DSP) and communications technology and is the reason why mp3 encoders and highly complicated multi-carrier radio transmitters are feasible with affordable equipment. Multi-carrier transmission, in the form of 'orthogonal frequency-domain multiplexing' (OFDM) has revolutionised and continues to revolutionise mobile communication systems, forms of which are the basis of fourth generation (4G) mobile communications. Therefore, study of the FFT is worthwhile. It is sometimes referred to as the Swiss army knife of DSP and communications.

Another transform commonly used for music and image processing is known as the Discrete Cosine transform (DCT). It has similar properties to the DFT and FFT, but manages to avoid complex numbers. The DCT is not the FFT with the imaginary part, or perhaps the phase, discarded. The DCT and inverse-DCT are available in Python (fftpack.dct and fftpack.idct) and can be used for frequency-domain processing in much the same way as the FFT and its inverse.

The DCT takes N real time-domain samples and produces N real frequency-domain coefficients. Each of these coefficients is a value of spectral energy for a frequency in the range 0 to *Fs*/2 Hz. The inverse-DCT converts the *N* real frequency-domain coefficients back to *N* real time-domain samples. The DCT does not produce mirrored frequency-domain samples as does the FFT. The frequencies corresponding to each DCT spectral sample are 0, *Fs*/(2*N*), 2*Fs*/(2*N*), …, *Fs*/2. Therefore, we get twice as many useful frequency-domain samples for *N* time-domain samples as we get using the FFT.

The FFT or DCT must be applied to segments of a signal that are, or may be considered 'stationary'. For such segments, the spectrum should not change significantly from beginning to end. If the sound is a musical note, it must be assumed not change, or even begin and/or end suddenly within the segment. Therefore we should not apply the FFT or DCT to long segments. Applying the FFT or DCT to say one minute or even one second of speech or music would generally be a mistake. Sounds that remain stationary for long periods of time would be profoundly boring, and it is the changes in speech and music that convey the meaning and the interest. There is a dilemma, and it is solved by assuming that long term signals are 'short-term stationary'. This simply means that if the segments are made short enough, the contents may be considered approximately stationary and processed as though the segment is exactly stationary. Speech and music segments of length less than about 20 ms can often be considered stationary. This means 160 samples, or less, for 8 kHz sampled speech and 882 samples or less for 44.1 kHz sampled speech or music.

The FFT and DCT are useful to enable the processing of signals in the frequency-domain. The inverse-FFT or inverse-DCT converts a spectrum back to the time-domain to allow the effect of any processing to be observed. For example, we could filter out certain frequency components by setting them to zero in the frequency-domain before converting back to the time-domain.

The objective of transform coding is to apply some processing to each spectrum before transmitting it. We could, for example, set to zero any frequency components that are unlikely to be heard at the receiver and therefore need not be sent. This is a good idea, though it introduces a problem. At the receiver, the processed segments may no longer join up perfectly and a nasty 'click' may be produced at the beginning of each new segment. A way of eliminating such 'clicks' is to use overlapping frames. Assuming each frame is of length *N*, we form extended frames, of length 2*N*, by combining each frame with the one before it. These extended frames are Hamming or Hann windowed and supplied to the FFT or DCT transform. At the receiver, each received frame is formed by adding the first half of the current received extended frame with the second half of the previous received extended frame. Of course, the previous received frame must be zeroed at the beginning of the decoder.

The FFT is essentially a Fourier series with fundamental frequency *Fs/N* Hz. It expresses any signal in terms of harmonics of *Fs/N* which are the frequency-domain sampling points. If we analyse a musical note or sine-wave whose frequency does not line up with one of these points, we do not just lose sight of the signal. Instead its energy is shared out among neighbouring frequency-domain sampling points. This sharing occurs with the DCT also. The way the energy is 'shared out' is improved by the use of non-rectangular widowing with, for example, a Hann or Hamming window as mentioned in Lectures.

Having read the theory above and the Week 3 Lectures, you may already have some idea how to design a transform coder. You may also be aware of some of the problems you will encounter. You should be in a reasonable position to solve these problems. Task 2 is broken down into the following components:

Part 2.1: Fourier series [3 marks]

Write Python programs to plot 500 samples of the periodic waveforms which have the following Fourier Series, where *F* = 500 Hz. Take the sampling frequency *Fs* to be 44.1 kHz.

$$(a)\ \mathrm{x(t)} = \sum_{k=0}^{\infty} \mathrm{B_k} \sin(2\pi kFt) \quad where\ \ B_k = \begin{cases} 0 & : \ k\ even \\ 1/k & : \ k\ odd \end{cases}$$

i.e. *x(t) = sin(2πFt) + (1/3) sin(2π(3F)t) + (1/5)sin(2π(5F)t) + (1/7)sin(2π(7F)t) +…*

$$(b)\ \mathrm{x(t)} = \sum_{k=0}^{\infty} \mathrm{A_k} \cos(2\pi kFt) \quad where\ \ A_k = \begin{cases} 0 & : \ k\ even \\ 1/k^2 & : \ k\ odd \end{cases}$$

i.e. *x(t) = cos(2πFt) + (1/9) cos(2π(3F)t) + (1/25)cos(2π(5F)t) + (1/49)cos(2π(7F)t) +…*

$$(c)\ \mathrm{x(t)} = \sum_{k=0}^{\infty} \mathrm{B_k} \sin(2\pi kFt + \phi_k) \quad where\ \ B_k = \begin{cases} 0 & : \ k\ even \\ 1/k & : \ k\ odd \end{cases} \qquad \phi_k = \begin{cases} \pi/2 : k = 3 \\ 0 : k \neq 3 \end{cases}$$

i.e. *x(t) = sin(2πFt) + (1/3)sin(2π(3F)t+π/2) + (1/5)sin(2π(5F)t) + (1/7)sin(2π(7F)t) +…*

Each of these Fourier series has an infinite number of terms, so you must truncate them.
Questions:
1. Comment on the waveforms obtained for about 4 or more non-zero harmonics.
2. What would you expect the waveforms to look like if you could take a very large number of harmonics?
3. In what ways are waveforms (a) and (c) similar and different?
4. In what ways are waveforms (a) and (b) related to each other?
5. Why might waveforms (a) and (c) sound similar over an analog telephone line?
6. If the waveform in (a) represented a sequence of pulses sent over an analog telephone line to represent a stream of bits, and the harmonics were affected by phase distortion to produce waveform (c), why might this cause a problem at the receiver?

Part 2.2: Frequency-domain processing [4 marks]

The file 'noisySinewave.wav' contains a characteristic periodic sound received from far away, and therefore contaminated by white noise. Write a Python program that reads in the wav file and plays it out using 'Audio'. Extract a segment of about 500 samples and plot the waveform as a graph of voltage against time. Then apply an FFT to obtain a complex array (with real and imaginary parts) with the same number of elements (say 500). You can plot the real and imaginary parts of this array if you wish, but they are usually difficult to understand. Instead, we usually convert each complex number to magnitude and phase form, and we often just concentrate on the magnitude. This gives us a 'magnitude spectrum' and each array element corresponds to a frequency.
Display graphs of the magnitude spectrum with the horizontal axis labeled firstly with array indices, and then with frequency values. Identify how (a) the periodic part of the sound and (b) the noise affect this magnitude spectrum. Decide on a suitable threshold and set to zero the real and imaginary parts of all the spectral samples whose magnitudes are below this threshold. Remember that the FFT of a real waveform is usually complex with 'symmetry' about half the sampling frequency. To be precise, the real part must be symmetric and the imaginary part anti-symmetric. You must preserve these properties when you apply frequency domain processing before performing the inverse FFT. Now perform an inverse-FFT and examine the resulting waveform. The FFT is documented in: http://docs.scipy.org/doc/numpy/reference/routines.fft.html
Questions:
1. If the original time-domain signal has N samples, how many frequency-domain samples do we obtain after performing the FFT? Why do we only need to plot the first N/2 samples of the magnitude spectrum?
2. How is periodic part of the sound and the noise seen in the magnitude spectrum?

3. How do you convert FFT frequency sample ('bin') number to frequency?
4. Would you describe the noise as being 'white'? Please explain your answer.
5. What value of threshold was chosen and why?
6. As you must apply the frequency-domain processing to the real and imaginary parts of the FFT individually, why would it be wrong to calculate thresholds for these parts individually rather than for the magnitude spectrum?
7. After applying the inverse FFT, is the resulting processed signal real, i.e. does it have zero imaginary part? If it were not real, what would you conclude from this?
8. Has any of the noise been removed by this 'spectral subtraction' process?
9. What Fourier series components are present in the periodic part of the sound?
10. Did you have any chance of answering question 9 by just observing the time-domain waveform (i.e. if you had never heard of the FFT)?

Hint: Remember that the FFT gives the spectrum at frequencies 0, *Fs/N*, 2*Fs/N*, …, *Fs* where *Fs* is the sampling freq and *N* is the number of samples. We normally plot only half the FFT magnitudes, that is up to *Fs*/2 (with index *N*/2), because of the Sampling Theorem.

Part 2.3: Transforming music files to & from frequency-domain [3 marks]

We can use either a FFT or a DCT. The FFT is familiar now, but it has the disadvantage of generating complex numbers. The DCT and inverse-DCT are available in Python (see fftpack) and can be used for frequency-domain-processing in much the same way as the FFT and its inverse. So let's use the DCT.

Test the DCT by generating a sine-wave segment of 1024 samples, sampled at *Fs* = 44.1 kHz, and applying it to the DCT. Plot its magnitude spectrum and check that you can discern the frequency of the sine-wave. Then apply the inverse-DCT, and if you get back to the original sine-wave, all is well. The DCT and its inverse are documented in: http://docs.scipy.org/doc/scipy/reference/fftpack.html

To make DCT work properly, use the following statements:

```
from scipy import fftpack
S = fftpack.dct(s, norm='ortho')
s = fftpack.idct(S, norm='ortho')
```

The 'ortho' is important here, in order to normalise the DCT.

Now read the file HQ-music44100-mono.wav of CD quality monophonic music into an int16 array, and process about 20 seconds of it in segments containing1024 samples. Each segment represents about 0.023 seconds of sound, so there will be about 870 segments. For each segment, apply a DCT to produce an array, 'dctF' say, convert its elements into 16-bit integer form (in the range -32768 to +32367) and store the resulting array, idctF, into a file. You may have to scale the elements of dctF to make sure they really do lie between -32768 and 32767. To save the array into a file, use either 'numpy.save' as documented in: http://docs.scipy.org/doc/numpy/reference/generated/numpy.save.html

or use:

```
with open ("filename.bin","wb") as f:
    f.write(dctF)
```

When you have processed all the segments, close the file. You will have a file containing a frequency-domain representation of the music. Re-open the 'binary' file, this time for reading. Read the frequency-domain data back into an array segment-by segment and check that you can reconstruct the original music without any distortion. Use 'Audio' to listen to the sound.

So far there is (almost) nothing lost and nothing gained in terms of bit-rate, but we do now have a frequency-domain representation of the music.

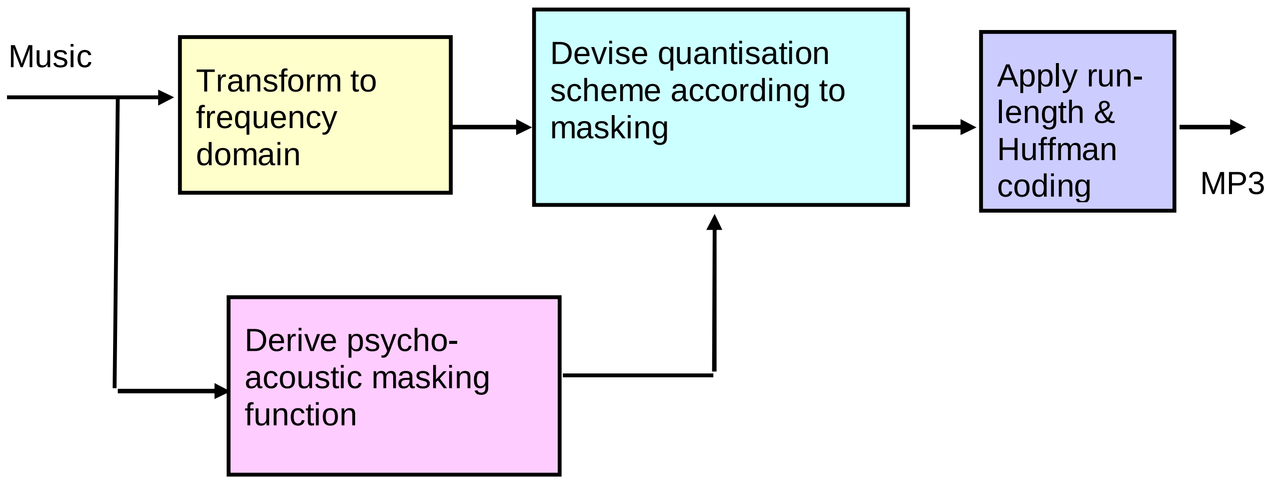
Questions
1: Why does the music have to be split up into sections when we wish to to apply frequency-domain processing?
2: Does the transformation, saving to a file, reading back from the file and/or reconstruction introduce any noticeable distortion?
3. If there was some distortion, what would be the cause of it?

Part 2.4 Principles of mp3 encoding [4 marks]

These principles of mp3 encoding are illustrated by the block-diagram below. The transform to the frequency-domain is often done as a 2-stage process, but we can apply a DCT directly as in Task 2.3.

We can reduce the bit-rate a little by not encoding any DCT coefficient above 16 kHz. Then we can derive a psycho-acoustical masking function and eliminate any DCT components that fall below it and would therefore not be heard. For DCT components above the masking threshold, the number of bits per DCT coefficient can be varied to achieve further bit-rate saving.
For the purposes of this experiment, we will use a psycho-acoustical masking function which remains constant for all frequencies. The effect of using more realistic masking functions will be demonstrated later.

For the constant masking function, you must define a threshold and set to zero all DCT coefficients whose magnitudes are less than this threshold. Experiment with this threshold to find the highest level for which reasonable music quality is obtained. This should produce many zero-valued DCT coefficients. Your program should produce sound bars (with titles) illustrating the effect on the sound, and the bit-rate saving obtained when using different thresholds. Also experiment with the number of bits per DCT coefficient.
Set your program to count the number and percentage of DCT coefficients that remain above the threshold for each frame. When all frames have been processed, your program should calculate and print out the overall percentage of non-zero DCT samples. The plan will be to avoid sending the zero values, but, for the moment, continue to store them as 16-bit integers (with perhaps not all of the bits used).


Music
Transform to frequency domain
Devise quantisation scheme according to masking
Apply run-length & Huffman coding
MP3
Derive psycho-acoustic masking function

Questions:
1. What saving in bit-rate can achieved by reducing the bandwidth to 16 kHz?
2. What is the effect of varying the constant threshold? What happens with a value that (a) is definitely too high, (b) too low and (c) potentially satisfactory? Give the values.
3. For the 'potentially satisfactory' threshold, what is the effect of quantising the remaining non-zero DCT coefficients to 8 bits per sample?
4. For the 'potentially satisfactory' threshold, and 8 bits per DCT coefficient, assuming

that you could send the zero-valued DCT coefficients at negligible cost, what bit-rate is required by this coding procedure? What bit-rate saving (in percentage terms) is achieved?

5. Did you observe any distortion in the form of 'clicks' due to discontinuities at frame boundaries?
6. If so, what causes these clicks? Think carefully about this and consider why you do not get clicks until you start doing some frequency domain processing.

Part 2.5. Simple method for eliminating discontinuities [2 marks]

The sound produced by Task 2.4 may be spoilt by nasty clicks caused by discontinuities at frame boundaries. These will not occur until processing is applied, and they become worse the more complex the processing is. Can you think of a way of 'smoothing over' the discontinuities at frame boundaries with some simple processing. Produce one sound bar to illustrate your simple method.

Questions

1. Describe the principle of your simple method and how it was implemented.
2. Does it work? If so how well does it work?

Part 2.6: Run-length & Huffman coding [2 marks]

We need to take advantage of the long strings of zero valued DCT coefficients that occur. This can be achieved by a form of 'run-length coding' where the number of zeros that precede or follow each non-zero coefficient is encoded as a separate number.

Take an example of a single DCT frame and devise a suitable run-length coding scheme. Assuming that each run-length of zeros is encoded using an 8-bit integer, estimate the bit-rate saving that is achieved in comparison to using 8-bit integers to encode all zero values. Do this for a single frame, then analyse the multiple frames for a file of music.

Question 1: What bit-rate saving was achieved for the single frame by run-length coding of zeros?

Question 2: What bit-rate saving was achieved for the whole file by run-length coding of zeros?

Huffman coding allows us to encode the remaining non-zero coefficients in a highly efficient way, taking into account how often each coefficient value will be expected to occur. Commonly occurring values are given shorter code-words than others, and each code-word is self terminating. Huffman coding will be considered in a future lecture.

Part 2.7. Eliminating discontinuities using overlapping frames [2 marks]

The preferred way of eliminating discontinuities is not so simple. It is to use overlapping frames. You can try this if you wish, but please do not spend too much time on it as we cannot give it all the marks it deserves.

When you analyse each new frame of length 1024 using the DCT, include the previous frame also to form a 2048 sample extended frame. Also impose a 2048 length Hann or Hamming window by multiplying the extended frame as follows:

```
winextFrame = extendedFrame * np.hamming(2048);
```

For documentation, see:
http://docs.scipy.org/doc/numpy/reference/generated/numpy.hamming.html
or
http://docs.scipy.org/doc/numpy/reference/generated/numpy.hanning.html#numpy.hanning

The windowed extended frame 'winextFrame' is now DCT transformed and processed as above, though there are now twice as many DCT coefficients to transmit.

At the receiver, we perform the inverse-DCT on each received array of 2048 DCT coefficients. We get 2048 time-domain samples, which is twice the number of samples that we need. But this allows us to overlap the frames and thus smooth out any discontinuities at frame boundaries. Sum the first half of the current received frame with the second half of the previous received frame. The array containing the previous frame must be set to zero once at the beginning of the decoder

Make this modification to the program written for Task 2.4 and remember to take into account the extended frame-length. Repeat the search for the best threshold and note whether the quality has improved with the overlapping frames. Again set your program to count the number of DCT coefficients that remain above the threshold, and note roughly how many such values there are per frame on average. Continue to store all samples, including zeros, as 16-bit integers.

Next, investigate whether we really need 16 bits per DCT coefficient. Continue to write 'int16' integers to the file, but try to discover how many of the least significant bits are not needed. Do this as in Task 1 by dividing and rounding to quantize the coefficients, and then multiplying by an integer (the one we divided by) to scale the volume back up (otherwise it is correct, but too quiet).

Questions:

1. With the best threshold for constant masking, how many non-zero DCT coefficients are there per frame on average?
2. How many bits per DCT coefficient did you find are really needed?
3. If we could find a way of sending the zero valued DCT coefficients at no cost (or very little cost), estimate the bit-rate saving that would result from the three techniques considered above.