6.003: Signal Processing

Short-Time Fourier Transforms

- Processing streaming signals
- Computational cost
- Overlap-add method

Quiz 2: November 2, 2-4pm, 50-340 (Walker)
  - Coverage up to and including all of week 7, including HW7.
  - Closed book except for two pages of notes (four sides total)
  - No electronic devices. (No headphones, cellphones, calculators, ...)
  - No HW8 – a practice quiz is posted.

October 26, 2021

Computational Cost

How many multiplies ($N_s$) are needed to implement a filter $h[n]$ with length $N_h=1024$ for a 3 minute song, sampled at 44,100 samples/second?

Compare three schemes:
  - direct convolution
  - using the DFT
  - using the FFT

Which is most efficient? Which is least efficient?

Computational Cost of Convolution

Find the number of multiplies required to compute the convolution $y[n] = \sum_{m=-\infty}^{\infty} h[m] x[n-m]$ where $x[n]$ is a 3 minute song sampled at 44,100 samples/second.

The total is approximately $3N_s^2 = 2 \times 10^{14}$ (>> $8 \times 10^9$ for convolution!)

Computational Cost of DFT-based Convolution

Find the number of multiplies required to compute the convolution with DFTs. To compute the DFT of the input $X[k] = \sum_{n=0}^{N-1} x[n] e^{-j 2\pi \frac{k}{N} n}$ requires $N$ multiplies for each value of $k$, which is $N^2$ in total.

This ignores the multiplies needed to compute the complex exponentials:
  - efficient algorithms exist for computing trig functions and
  - these coefficients can often be reused in subsequent calculations

Even though $N_h < N_s$, we need to compute $H[k]$ with the same resolution as $X[k]$ in order to multiply $H[k]$ times $X[k]$.

Therefore the total number of multiplies is:
  - $N^2$ to find $X[k]$ for all $k$,
  - $N^2$ to find $H[k]$ for the same values of $k$,
  - $N_s$ to multiply $H[k]$ times $X[k]$ for all $k$, and
  - $N^2_s$ to find $y[n]$ from $Y[k]$.

The total is approximately $3N_s^2 = 2 \times 10^{14}$ (>> $8 \times 10^9$ for convolution!)

Computational Cost of FFT-based Convolution

The Fast Fourier Transform (FFT) is a highly efficient algorithm for computing the DFT. The FFT requires approximately $N \log_2 N$ multiplies (instead of $N^2$) to compute a DFT of length $N$.

Therefore the total number of multiplies using the FFT is:
  - $N_s \log_2 N_s$ to find $X[k]$ for all $k$,
  - $N_h \log_2 N_h$ to find $H[k]$ for the same values of $k$,
  - $N_s$ to multiply $H[k]$ times $X[k]$ for all $k$, and
  - $N_s \log_2 N_s$ to find $y[n]$ from $Y[k]$.

The total is approximately $3N_s \log_2 N_s = 6 \times 10^8$

method                   number of multiplies
  direct convolution      $8 \times 10^9$
  DFT                     $2 \times 10^{14}$
  FFT                     $6 \times 10^8$

FFT is more than 300,000x faster than the DFT!

Streaming Music

One of the most important attributes of the DFT is that it can be computed from a finite part of a potentially very long signal.

The DFT is defined for $0 \leq n < N$ while the DTFT has infinite limits.

$$DFT: \quad X[k] = \sum_{n=0}^{N-1} x[n] e^{-j 2\pi \frac{k}{N} n}$$

$$DTFT: \quad X(\Omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-j 2\pi \Omega n}$$

Breaking a signal into pieces so that it can be processed in small chunks is especially important in streaming applications where the signal may be arbitrarily long.

Breaking a signal into pieces also affects the number of computations required to perform a signal processing task.
Computational Cost
It is easy to see why the FFT requires fewer multiplies than the DFT since the number of multiplies goes as $N \log_2 N$ instead of $N^2$.

And there is still room for improvement!

Neither of the Fourier based schemes took advantage of the fact that the filter length $N_h=1024$ is small compared to that of the signal $N_s = 8 \times 10^6$.

We can improve the performance of the Fourier methods by using a short-time Fourier method, which allows us to use smaller transforms that are better matched to $h[n]$.

## Short-Time Fourier Transforms
Short-time Fourier transforms are based on the analysis of a sequence of finite-length portions of an input signal.

General procedure:
- Divide the input signal into a sequence of windows, each of length $N$.
- Process each window.
- Assemble the processed pieces together to form the output.

## Example
Consider a musical piece that contains three simultaneous “voices,” each playing a single sinusoidal tone (lab 7):

<table>
<thead>
<tr>
<th>Voice</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (bass)</td>
<td>40-170 Hz</td>
</tr>
<tr>
<td>2 (melody)</td>
<td>170-340 Hz</td>
</tr>
<tr>
<td>3 (harmony)</td>
<td>340-750 Hz</td>
</tr>
</tbody>
</table>

We would like to remove the bass and harmony voices, leaving just melody.

## Algorithm 1
There are at least two major problems with this approach.

- The length of $(x \ast h)[n]$ is generally > length of $x[n]$ or $h[n]$.
  - Part of result from each window should fall into an adjacent window(s).
- Even worse, the convolution will be circular if implemented with a DFT.
  - Results from window 1 that should fall into window 2 will alias back to the beginning of window 1!

## Overlap-Add Method
Avoid circular convolution artifacts and spill over problems by filling each window with just $s < N$ input samples and then zero-padding.

If the length of $h[n]$ ≤ $N-s+1$, then the length of $h[n]$ convolved with $s$ samples of the input will be less than $N$ → no circular convolution artifact.

If the length of $h[n]$ ≤ $N-s+1$, then the overlapping portions of adjacent windows will accomodate spill over between windows.
**Overlap-Add: Graphical Depiction**

Convolving a square pulse with a signal that is 1 for all n.

\[ h[n] \]
\[ x[n] \]

Divide the input \( x[n] \) into pieces that are each of length \( s \).

\[ x_0[n] \]
\[ x_1[n] \]
\[ x_2[n] \]

Convolve each piece of \( x[n] \) with \( h[n] \).

\[ y_0[n] \]
\[ y_1[n] \]
\[ y_2[n] \]

Then the output \( y[n] = y_0[n] + y_1[n] + y_2[n] + \cdots \) Hence overlap-add.

**Filter Design**

How can we design a filter with 2048 points in time (\( n \)) but 8192 points in frequency (\( k \))?  
- Start by designing a filter \( H_1[k] \) with length \( N_f = 2048 \). The filter should only pass frequencies in the range \( f_l \leq f \leq f_h \).  
- Convert \( H_1[k] \) to the time domain using an inverse DFT. The length of the resulting \( h_1[n] \) will be \( N_f = 2048 \).  
- Define a new filter \( h_2[n] \) which is a version of \( h_1[n] \) that is zero-padded to a new length of \( N = 8192 \).  
- Convert \( h_2[n] \) to the frequency domain to get \( H_2[k] \).

The filter \( H_2[k] \) will have 8192 values of \( k \) but its time-domain representation \( h_2[n] \) will have just 2048 non-zero values.

**Filter Design**

Design a filter to isolate the melody using the overlap-add method.

The filter should pass frequencies in the range \( f_l \leq f \leq f_h \).

\[ k_i = \frac{k_i}{f_s} \]
\[ k_h = \frac{k_h}{f_s} \]

If we take the window length \( N = 8192 \), then \( h[n] \) has that same length.

**Filter Design**

Zero-pad to make filter length equal to window length \( N = 8192 \).

Listen to result: `am_filtered.wav`  
Significant improvement. No clicks. Bass and harmony are faintly audible — probably because of deviations from ideal filters. Ripples are due to Gibb’s phenomenon.
Gibb’s Phenomenon
Ripples in frequency result from windowing in time.

A rectangular window in time
\[ w_r[n] = \begin{cases} 1 & \text{if } 0 \leq n < N \\ 0 & \text{otherwise} \end{cases} \]
corresponds to a DT sinc in frequency.

\[ W_r(\Omega) = \sum_{n=-\infty}^{\infty} w_r[n] e^{-j \Omega n} = \sum_{n=0}^{N-1} e^{-j \Omega n} = \frac{1 - e^{-j \Omega N}}{1 - e^{-j \Omega}} = \frac{\sin(\frac{\Omega N}{2})}{\sin(\frac{\Omega}{2})} e^{-j \Omega (N/2)} \]

Multiplying the unit sample response \( h[n] \) by a window function, convolves the desired bandpass shape with the DT sinc – generating ripples.

Filter Design
We can reduce the passband ripple by applying a triangular window (red).

A triangular window in time
\[ w_t[n] = \sin\left(\frac{\pi n}{N}\right)^2 \]
corresponds to a DT sinc squared in frequency.

\[ W_t(\Omega) = W_r^2(\Omega) \]

\( H_2[k] \) is now a smoother function of \( k \), rippling is greatly reduced.
Listen to result: am_triangular.wav

Gibb’s Phenomenon
Ripples in frequency result from windowing in time.

A Hann window in time
\[ w_h[n] = \sin\left(\frac{\pi n}{N}\right) \]
produces even smaller ripples.

\[ W_h(\Omega) = \frac{\sin(\frac{\Omega N}{2})}{\sin(\frac{\Omega}{2})} e^{-j \Omega (N/2)} \]

Better yet, try a Hann window (red).

Filter Design
We can reduce the passband ripple by applying a triangular window.
**Filter Design**

Better yet, try a Hann window.

\[
h_2[n] = \begin{cases} 
1 & \text{if } |n| < \frac{N}{2} \\
\frac{1}{2}(1 - \cos \frac{\pi n}{N}) & \text{otherwise}
\end{cases}
\]

\[
H_2[k] = \frac{1}{N} \sum_{n=-N/2}^{N/2} h_2[n] e^{-j2\pi nk/N}
\]

\[
H_2[k] \text{ is now even smoother.}
\]

Listen to result: am_Hann.wav

---

**Computational Cost of Overlap-Add Method**

Does breaking signal into windows increase or decrease computational cost?

Each FFT of length \( N \) contributes \( s \) samples to the output.

\[
N \quad N \quad N
\]

Window 0

Window 1

Window 2

\[
N \quad N
\]

\[
-128 \quad -64 \quad -32 \quad 0 \quad 32 \quad 64 \quad 128
\]

\[
H_2[k] \quad k \text{ (expanded scale)}
\]

The total number of multiplies \( N_t \) to process a long signal (\( N_x \) samples) is the number of windows (\( N_x/s \)) times the number of multiplies per window (\( 2N \log_2 N \), since we only need to calculate frequency response once).

For today's example, \( N_x = 8 \times 10^6 \), \( s = 6144 \), \( N = 8192 \) so

\[
N_t = \frac{8 \times 10^6 \times 2 \times 6144 \times \log_2(8192)}{6144} \approx 3 \times 10^8
\]

which is only half as many multiplies for full-length FFTs.

Even more importantly, we can process the first window without waiting for the entire song to be transmitted – very important for streaming.

---

**Summary**

Breaking a signal into pieces so that it can be processed in small chunks is important in streaming applications where the signal may be arbitrarily long.

Breaking a signal into pieces affects the number of computations required to perform a signal processing task.

- Implementing convolution with an FFT can be much more efficient than direct convolution.
- Implementing convolution with short-time Fourier transforms based on the FFT can be similarly efficient.

When processing a signal in chunks, care must be taken to avoid introducing artifacts between chunks.

There are a number of signal processing schemes (such as overlap-add) to enable seamless stitching of separately processed windows.