Filtering With DFTs
Filtering (bass-boost, treble boost, equalization) is built into many modern loudspeaker systems. How much computation is needed? How many multiplies \( (N_m) \) are needed to implement a filter \( h[n] \) with length \( N_h=1024 \) on a 3 minute song, sampled at 44,100 samples/second?

Compare three schemes:
- direct-form convolution?
- with a DFT?
- with an FFT?

Which is most efficient? Which is least efficient?
Filtering Music

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

Start by finding the number of samples \(N_x\) in a song:

\[
N_x = 3 \text{ min.} \times 60 \text{ sec./min.} \times 44,100 \text{ samples/sec.} \approx 8 \times 10^6 \text{ samples}
\]

**direct-form convolution:** \(N_m = N_x \times N_h \approx 8 \times 10^9\)

(must sum \(h[m]x[n−m]\) over \(M\) for each sample)

**DFT:** \(N_m = 3N_x^2 + N_x \approx 2 \times 10^{14}\)

\((N_x^2\) to find \(X[k]\), \(N_x^2\) for \(H[k]\), \(N_x\) for \(Y[k] = X[k]H[k]\), and \(N_x^2\) for \(y[n]\))

**FFT:** \(N_m = 3N_x \log_2(N_x) + N_x \approx 5 \times 10^8\)

\((N_x \log_2(N_x)\) to find \(X[k]\), \(N_x \log_2(N_x)\) for \(H[k]\), \(N_x\) for \(Y[k] = X[k]H[k]\), and \(N_x \log_2(N_x)\) for \(y[n]\))

FFT is 16\(x\) faster than convolution and 400,000\(x\) faster than DFT!
Filtering Music

The FFT is faster than the DFT because the number of multiplies goes like $N \log_2(N)$ rather than $N^2$.

Convolution also goes as $N^2$. Why is DFT so much worse than convolution?

The convolution method takes advantage of the fact that the filter length $N_h=1024$ is small compared to that of the signal $N_x = 8 \times 10^6$. The Fourier methods did not. They could implement filters as long as the signal with no additional multiplies!

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 $h[n]$. 
Algorithm 1

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

Consider a musical piece that contains three parts:

\[ X(f) \]
\[
\begin{array}{c|c|c|c|}
\text{bass} & \text{melody} & \text{harmony} \\
\end{array}
\]

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

\[ H(f) \]
\[
\begin{array}{c|c|c|c|}
\text{} & \text{} & \text{} & \text{} \\
\end{array}
\]
**Algorithm 1**

Chop the input signal into pieces that are each of length $N$.

Filter each piece by zeroing FFT components outside passband.

Compare original `am_resynth.wav` to result `am_algorithm1.wav`.

How effective is this algorithm?
Algorithm 1

Chop the input signal into pieces that are each of length $N$.
Filter each piece by zeroing FFT components outside passband.

Q: How effective is this algorithm?
A: Not very.

One major problem with this algorithm is that if you convolve window 0 with a filter, part of the result should fall into window 1. This is not possible with algorithm 1.
Overlap-Add Method

Algorithm 1’s big problem can be fixed with overlapping windows.

How does overlapping help? How would you choose \( s \) and \( N \)?
Overlap-Add Method

How does overlapping help? How would you choose $s$ and $N$?

![Diagram of windows](image)

Fill each window with $s$ samples of the input and $N-s$ zeros.

Then convolve each window with the filter and sum the windows.

Notice that the convolution of the filter with $x[0:s]$ must not fall outside $0 \leq n < N$. If it did, it would wrap around to the beginning of window 0.
Convolve a square pulse with a signal that is 1 for all $n$. 

$h[n]$

$x[n]$

$x_0[n]$

$x_1[n]$

$x_2[n]$

$y_0[n]$

$y_1[n]$

$y_2[n]$
Filter Design

Design a filter for the overlap-add method: \( s = 6144 \) and \( N = 8192 \). The filter should pass frequencies in the range \( \Omega_l < \Omega < \Omega_h \).

Method 1: \( N = 8192 \)

\[
X[k] = \begin{cases} 
1 & \text{if } N \frac{\Omega_l}{2\pi} \leq |k| \leq N \frac{\Omega_h}{2\pi} \\
0 & \text{otherwise}
\end{cases}
\]

Method 2: \( N = 2048 \)

\[
X[k] = \begin{cases} 
1 & \text{if } N \frac{\Omega_l}{2\pi} \leq |k| \leq N \frac{\Omega_h}{2\pi} \\
0 & \text{otherwise}
\end{cases}
\]

Method 3: Start with method 2. Then take inverse FFT; zero-pad to \( N = 8192 \), and take FFT.

Method 4: Start with method 1. Then take inverse FFT, apply rectangular window with width 2048, and take FFT.
Design a filter for the overlap-add method: \( s = 6144 \) and \( N = 8192 \). The filter should pass frequencies in the range \( \Omega_l < \Omega < \Omega_h \).

Ultimately we need a filter \( H[k] \) of length \( N = 8192 \) (window size). However, \( h[n] \) must be no longer than \( N = 2048 \) samples.

Therefore, design a filter using \( N = 2048 \). Take the inverse transform. Pad to \( N = 8192 \) samples. Take the transform.

→ Could use method 3 or 4 (they are equivalent).
Filter Design

Design a bandpass filter to extract 170-340 Hz frequency region from signal sampled with $f_s = 44,100$ Hz with $N_f = 2048$.

\[
\frac{170}{f_s} \times N \approx 8 \leq k \leq \frac{340}{f_s} \times N \approx 16
\]

\[
H_1[k] \\
-\frac{N_f}{2} \quad 0 \quad \frac{N_f}{2}
\]

\[
h_1[n] \\
0 \quad \frac{N_f}{2}
\]
Filter Design

Zero-pad to make filter length equal to window length.

Listen to result: am_filtered.wav
What’s wrong with Method 3 (and 4)? Gibb’s phenomenon!

Fix: substitute a better window.

Method 5 (based on Method 1):

Start with method 1.
Then take inverse FFT and apply a higher-order window (such as a triangular window) of length 2048 instead of a rectangular window of length 2048.
Finally, take an FFT to find the $H[k]$. 
Filter Design

Apply a triangular window $w[n]$.

Notice that $H_2[k]$ is now a smoother function of $k$.

Listen to result: am_triangular.wav
Better yet, try a Hann window.

\[ h_2[n] \]

\[ H_2[k] \]

\[ H_2[k] \] is now even smoother.

Listen to result: am_Hann.wav
Overlap-Add Method

\[ H = \text{fft}(h2) \]

\[ N = 8192 \]
\[ s = 8192-2048 \]
\[ x, fs = \text{wav\_read('am\_resynth.wav')} \]

\[ x += N * [0] \quad \# \text{extend } x \text{ so last sample falls in last full-length window} \]
\[ \text{out} = [] \]
\[ \text{keep} = (N-s) * [0] \quad \# \text{for overlap that adds with next window} \]

for \( i \) in range(len(x) // s):
    \[ X = \text{fft}(x[i\ast s:i\ast s+s]+(N-s)*[0]) \]
    \[ Y = [x\ast h \text{ for } x, h \text{ in zip}(X, H)] \]
    \[ y = [y\cdot \text{real} \text{ for } y \text{ in ifft}(Y)] \]
    \[ \text{out}.\text{extend}([kk+yy \text{ for } kk, yy \text{ in zip}(\text{keep}, y[0:len(keep)])]) \]
    \[ \text{out}.\text{extend}(y[len(keep):s]) \]
    \[ \text{keep} = y[s:] \]

\text{wav\_write(out, fs, 'out.wav')}
Each FFT of length $N$ contributes $s$ samples to the output.

Number of windows $= N_x/s$.

Number of multiplies per window $\approx 2N \log_2(N)$
(only need to calculate frequency response once)

Total number of multiplies $\approx 2N_x \frac{N}{s} \log_2(N)$.

Typically $\frac{N}{s}$ is near 1 (it was $\frac{3}{4}$ in today’s example).

Total $\approx 2N_x \log_2(N)$.

Compared to $\approx 3N_x \log_2(N_x)$ 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 applications**.