Phase Vocoder Effects

Mutation, Robotisaton, Whisperisation,
Pitch shifting, Time scaling

Phase Vocoder

- Sound analysis-synthesis techniques performed in the spectral domain
  - widely recognised ‘standard’ implementation
- decompose signal over short windowed frames (STFT) to analyse frequency content over time
  - Take into account phase information
- Applications
  - Time scaling / pitch shifting of audio
  - Signal Content Analysis
  - Denoising
  - Sound synthesis by example
  - Pitch detection
  - Steady State/Transient Separation
  - …

Introduction: FFT

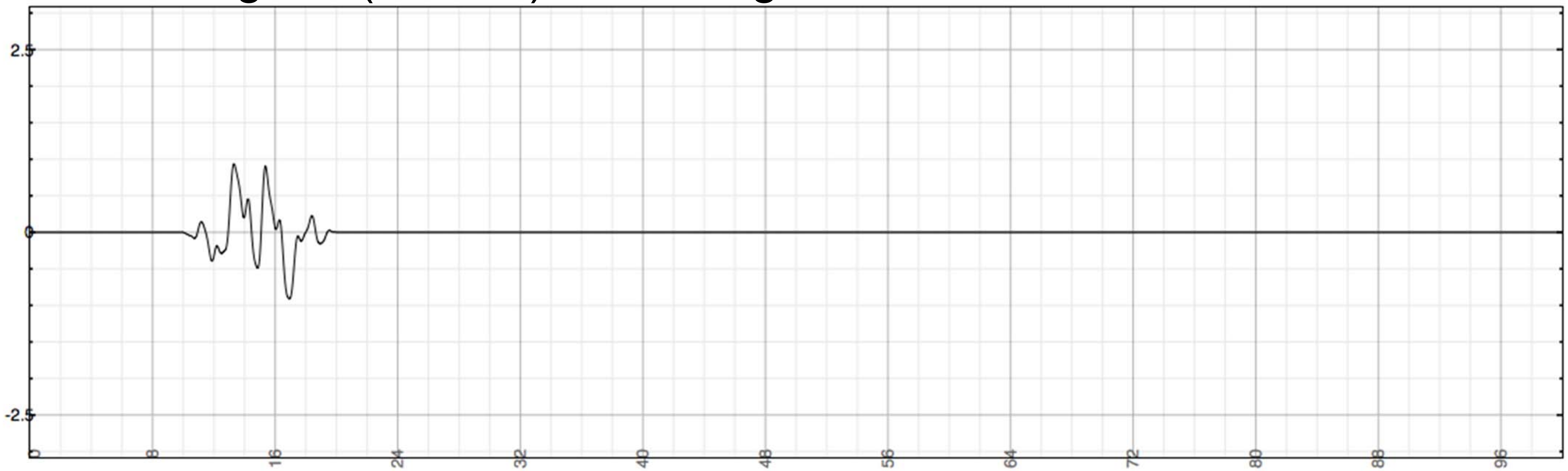
- What is the FFT?
  - Fast Fourier Transform
  - Algorithm to compute Discrete Fourier Transform (DFT) for evenly-spaced frequency samples
  - DFT can be seen as sampling of Discrete-Time Fourier Transform (DTFT)
    - DTFT: $$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}$$ (continuous frequency)
    - DFT: $$X_k = \sum_{n=0}^{N-1} x[n]e^{-j2\pi \frac{k}{N} n}$$ (discrete frequency)
    - FFT: Same as DFT: just a specific implementation

Introduction: FFT

- Reconstructing signal from DFT/FFT
  - Inverse Discrete Fourier Transform (IDFT/IFFT)
    - DFT: $$X_k = \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{k}{N} n}$$
    - IDFT: $$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{+j2\pi \frac{k}{N} n}$$
- Notice: N time samples $\longleftrightarrow$ N frequency samples
  - Two different perspectives on same signal
  - Given a signal of length M, can exactly reconstruct from an FFT of length N ≥ M
    - What happens with an FFT of length N < M? Aliasing

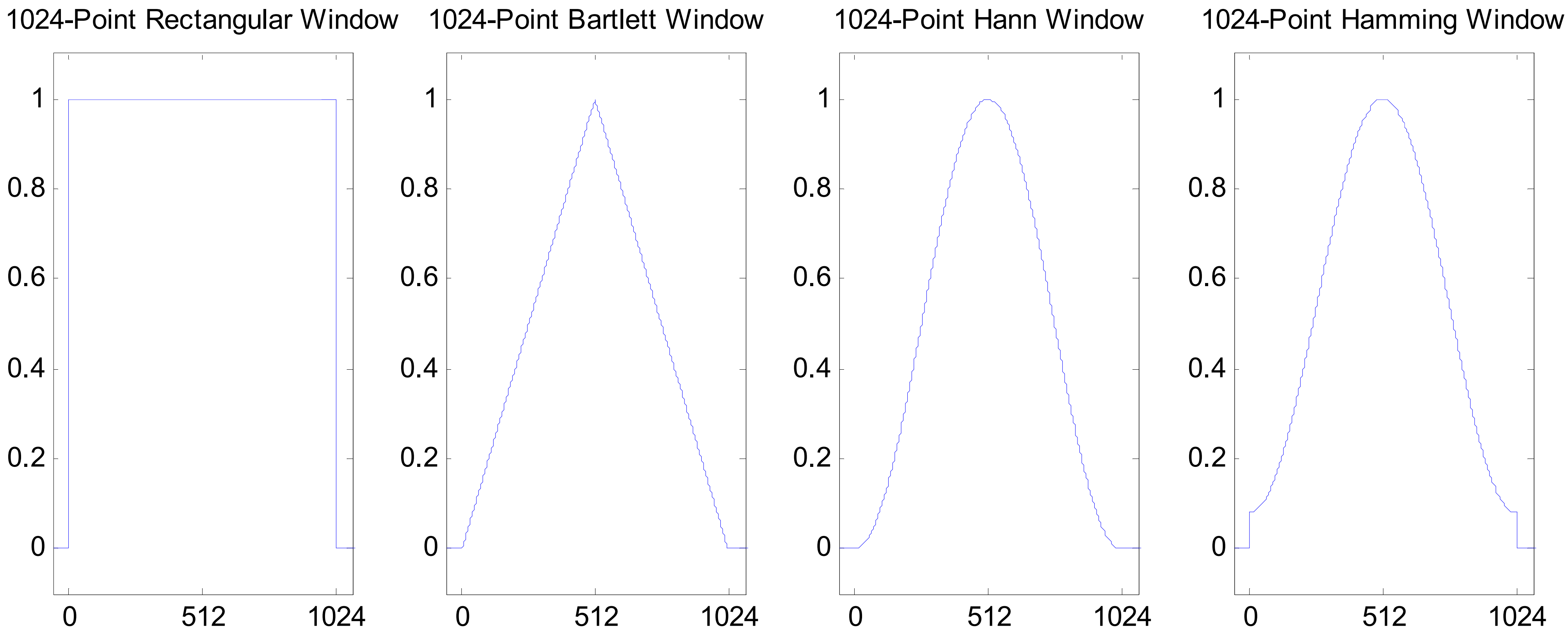
Windowing

- Suppose we look at only short slice of input signal
  - Want instantaneous snapshot of frequency content
  - Window the signal, then apply the DFT
    - Multiply by a function that's nonzero for a fixed length M
    - Might also have a shape within the nonzero section: rectangular, triangular (Bartlett), Hamming, Blackman, etc.

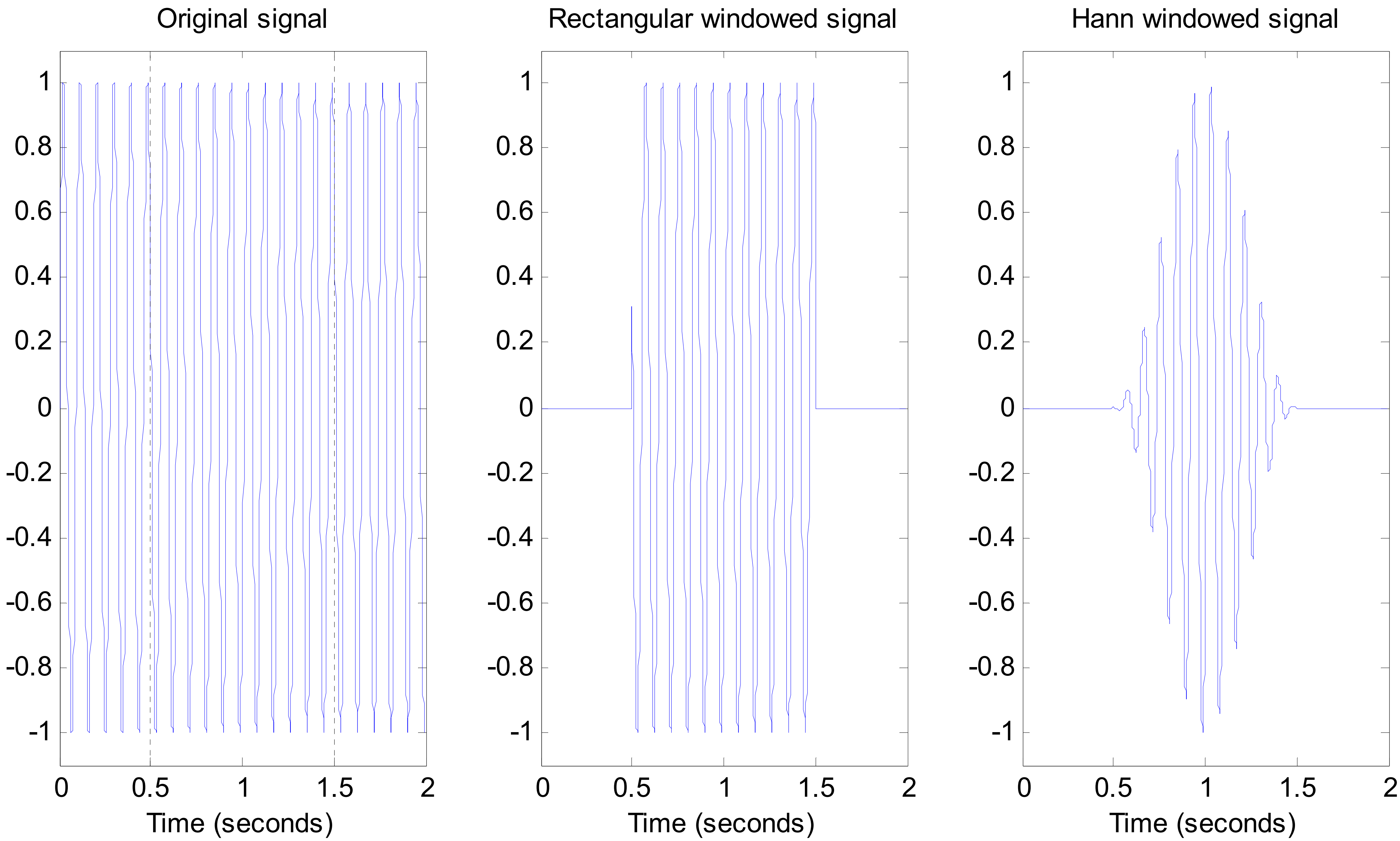


    - Can reconstruct windowed signal from DFT length N as long as $N \geq M$

Four popular window functions

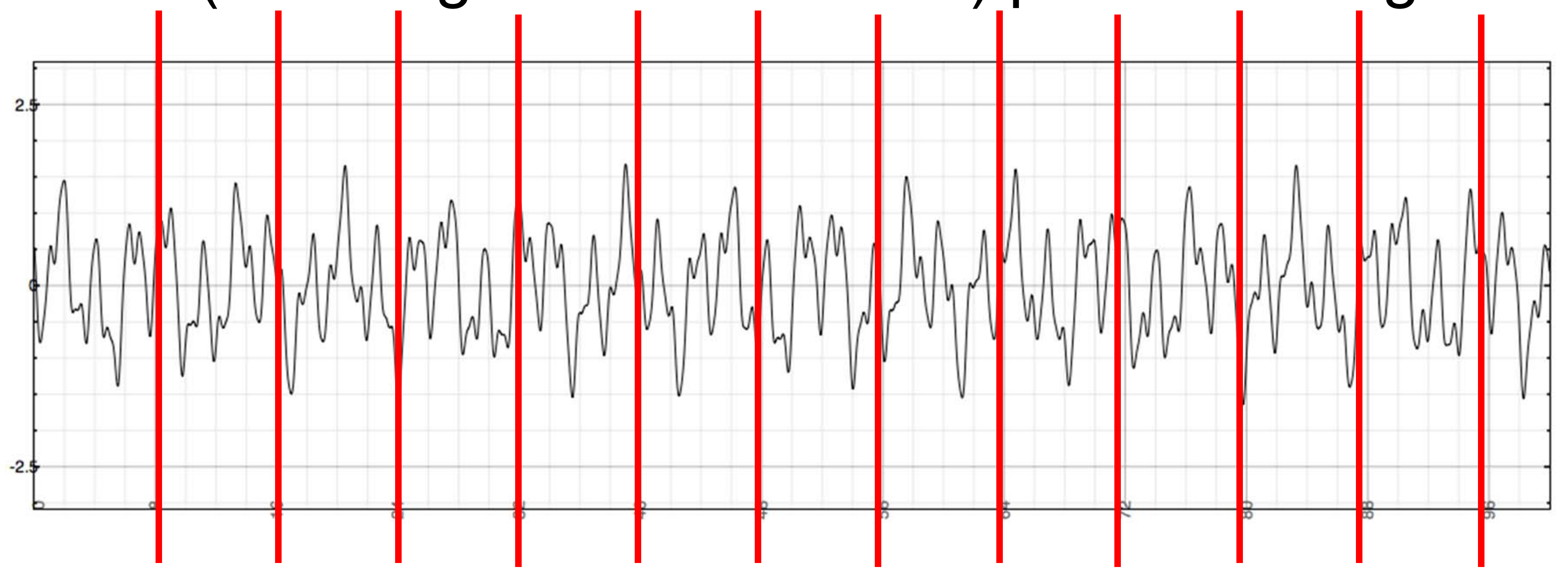


Effect of a rectangular window and a Hann window applied to a signal



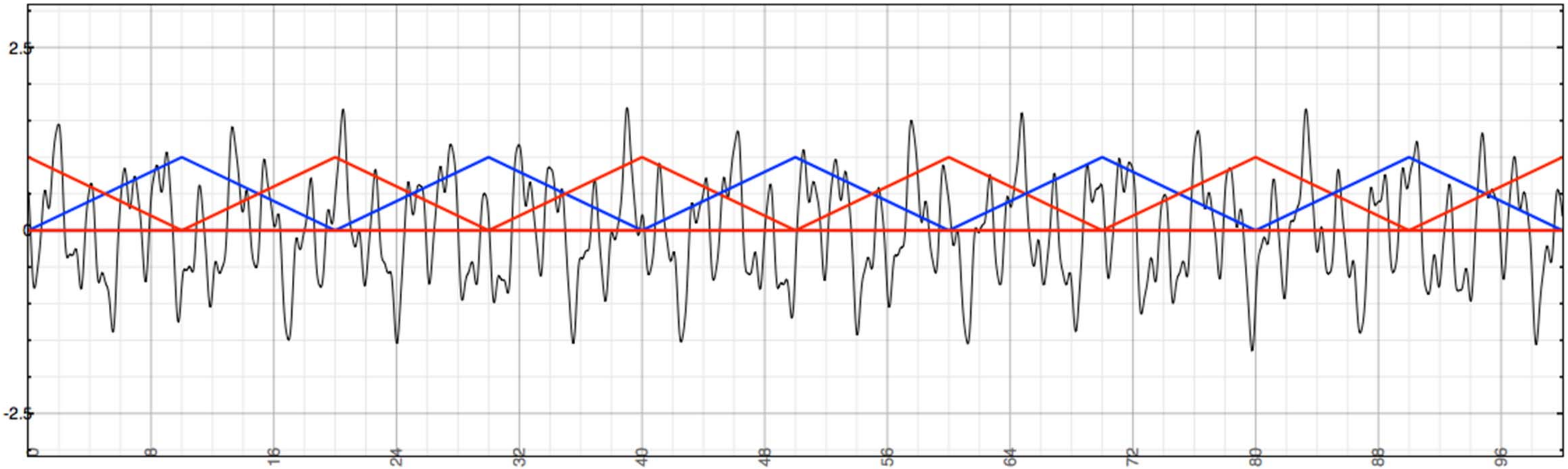
Short-Time Fourier Transform

- Short-Time Fourier Transform (STFT) is the DTFT of the windowed signal
  - $$\mathbf{STFT}\{x[n]\} \equiv X(m, e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n}$$
  - Similarly, can take DFT/FFT for discrete samples of windowed signals
- Can break any signal into windowed segments
  - ...then (in the right circumstances) put it back together



Overlap-Add

- Can split signal into overlapping segments
  - Distance between segments is called hop size
  - Here, reconstruction from triangular windows with hop size M/2 (can do same with Hamming windows)

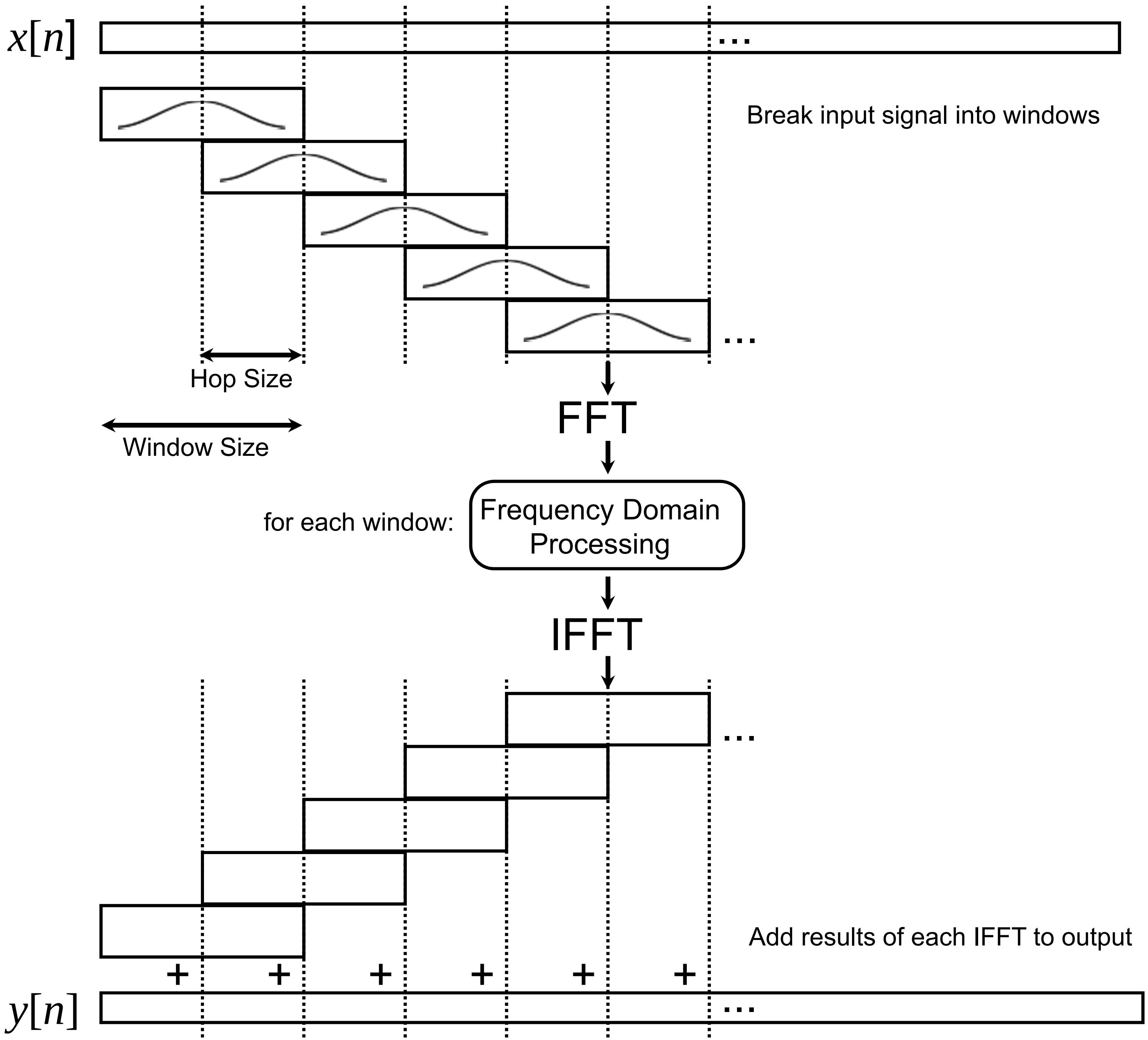


- Remember, take DFT of each segment to get frequency content, IDFT to get back...

Overlap-Add

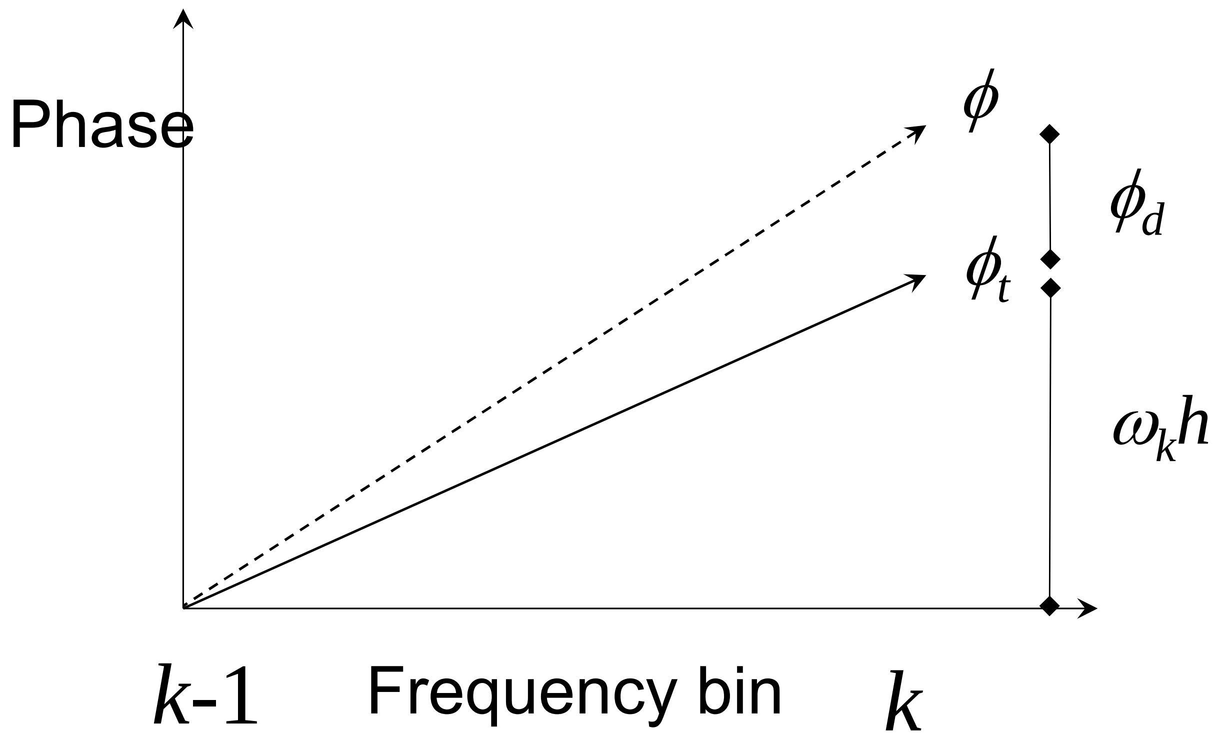
- Overlap-Add processing method:
  1. Take m^{th} segment (frame) of length M using windowing function
  2. Take DFT of length $N \geq M$ of segment
     - If $N > M$, zero-pad the segment (add zeros to end)
  3. Do something interesting to frequency data
  4. Take IDFT to get back to time domain segment
  5. Add result to output buffer containing prior segments
  6. Advance by hop size to $(m+1)^{th}$ frame and repeat

Overview of Overlap and Add Process



- Segment audio by applying window function
- Apply FFT to each segment
- Extract phases and amplitudes
- Do stuff!
- Apply IFFT to each segment
- Overlap segments and append on to output audio

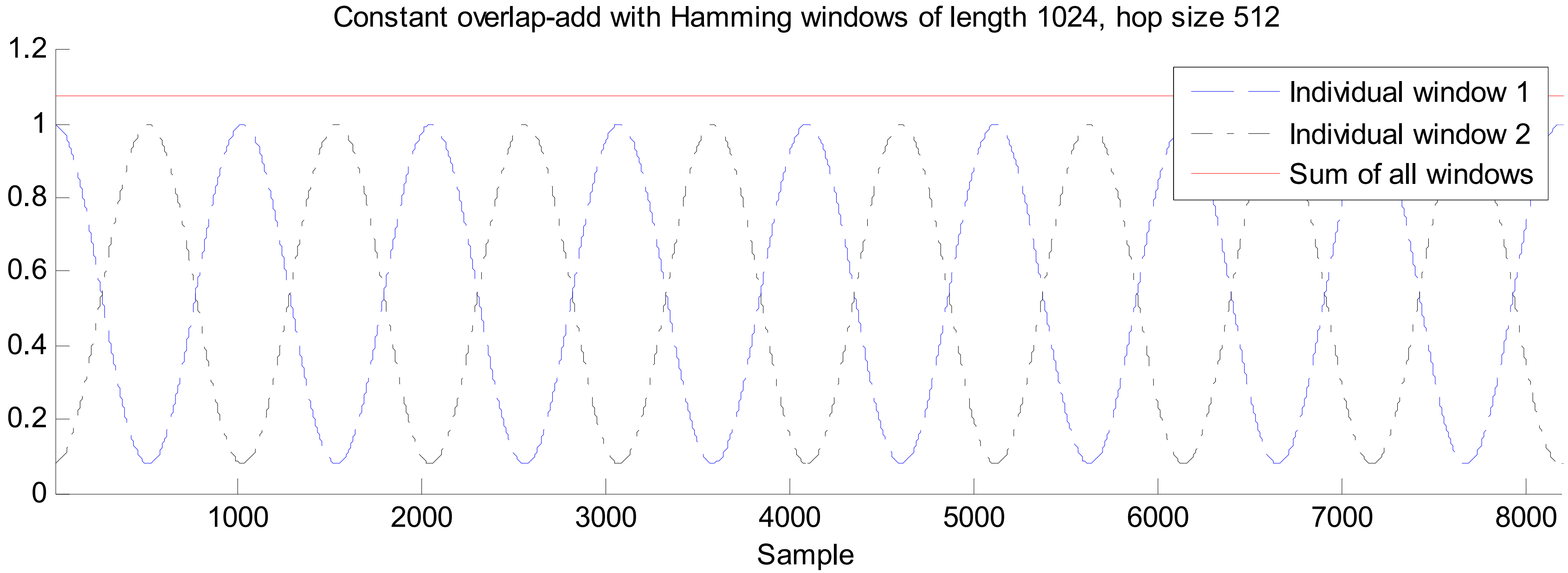
Relationship between actual phase and target phase.

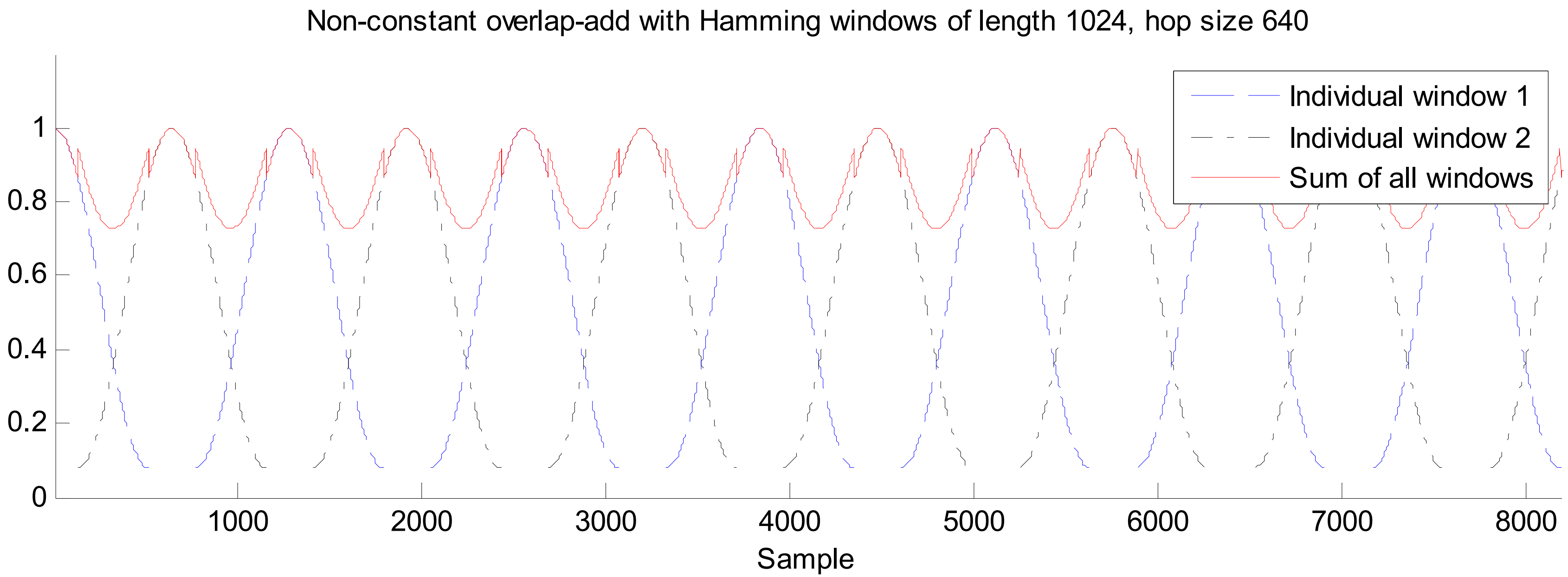


Instantaneous frequency represented by gradient of dashed line

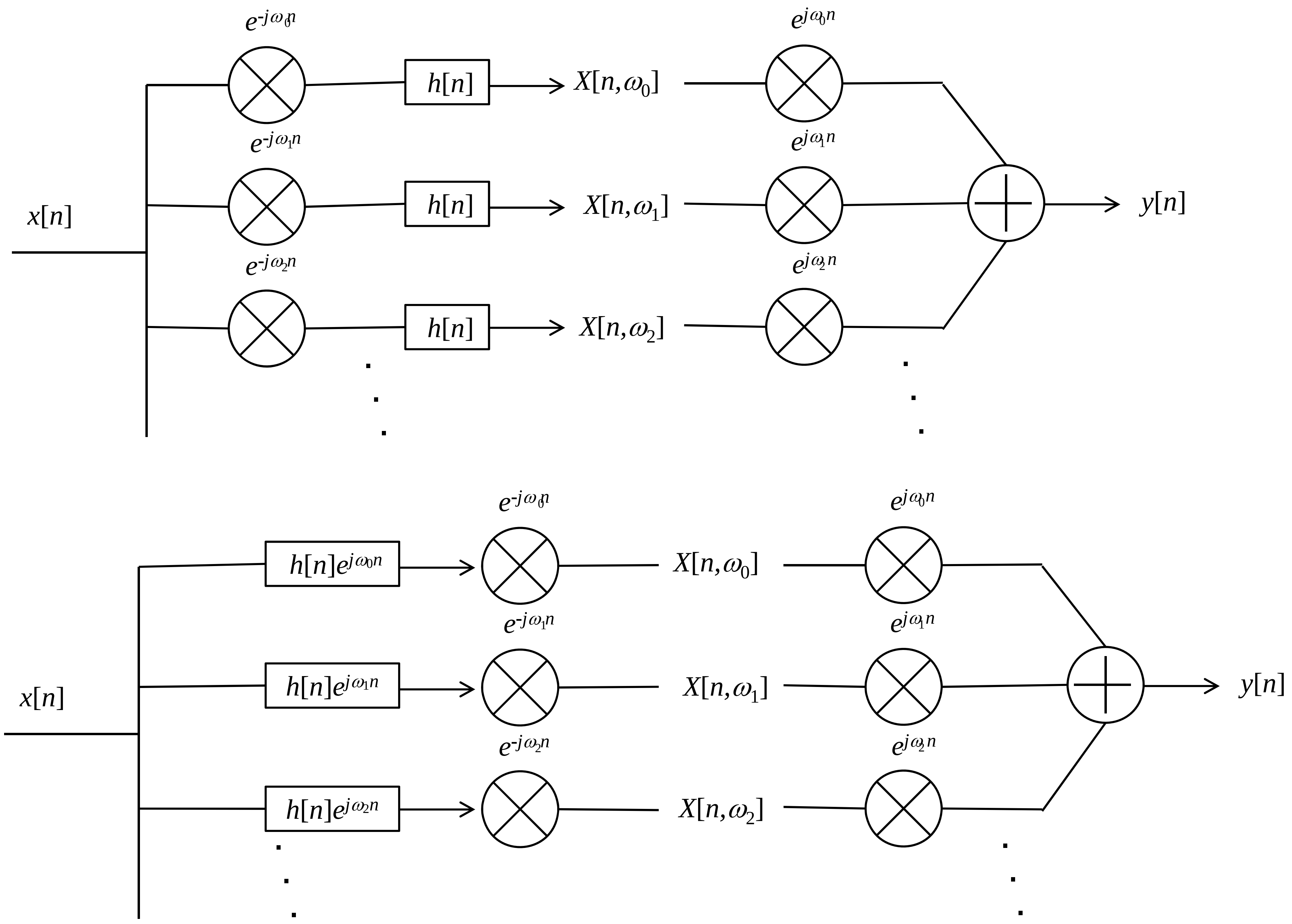
Bin frequency is gradient of solid line.

constant overlap-add criterion requires that window functions, when overlapped, add to constant value. holds in top plot, but not bottom plot.





complex baseband filterbank implementation (top) where h[n] is low pass filter, and complex bandpass implementation (bottom). Only filters for first 3 bins shown



Applications

- Many possibilities between DFT and IDFT
  - Efficient FIR convolution
    - FIR filter of length *N* needs *N* multiplies per sample
    - Convolution in time = multiplication in frequency

$$X(z) \longrightarrow \boxed{H(z)} \longrightarrow Y(z)$$

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

$$Y(z) = H(z)X(z)$$

  - Mutation and cross-synthesis
  - Robotisation and whisperisation
  - Time-stretching and pitch-shifting

Mutation

- DFT bins have magnitude and phase

$$z = x + jy \qquad |z| = \sqrt{x^2 + y^2} \qquad \arg(z) = \theta = \tan^{-1}\left(\frac{y}{x}\right)$$

- Suppose we have DFTs of two signals
  - Combine magnitudes and phases in interesting combinations
  - Sometimes known as cross-synthesis or morphing
  - For example, magnitude from one, phase from other

Mutation

- Example operations on magnitude only
  - Multiplication: $r = r_1 * r_2$
    - Multiplication in linear terms = addition in dB scale
    - Similar to spectral AND operation, keeping zones where energy is located in both signals
    - For fixed r_2 (and zero phase z_2), equivalent to FIR filtering
  - Addition: $r = r_1 + r_2$
    - Similar to spectral OR operation
    - Not simple mixing: only operating on magnitudes, not phases
  - Masking: $r = (r_2 > \text{threshold} ~?~ r_1 : 0)$
    - Keep magnitude of one sound only if the other is above a threshold
    - (A ? B : C) syntax means “if A, then B, otherwise C”

Mutation

- Phase is critical to time-frequency representations
  - Explains how signals evolve from frame to frame
  - Has an important influence on output quality
- Example operations on phase of two inputs
  - Keep phase of only one sound, change magnitude
    - Phase is a strong cue for pitch of sound (why?)
    - Result: pitch influenced by the phase you keep
  - Add phases
    - Strong alteration: phase moves at double speed on average
    - Or double the reconstruction hop size: $n_2 = 2 * n_1$
  - Arbitrary combination or variation on phases

Robotisation

- Robotisation = applying fixed pitch onto a sound
  - Implemented by setting DFT phase values to 0 before reconstruction
  - Forces periodicity: erratic and random variations converted into fixed-pitch sound
  - Pitch determined by hop size between segments
    - Can even be adjusted dynamically to create pitch changes
  - “Robot voice” effect

Robotisation VST I

```
int inwritepos;         // Write pointer into the input buffer
int outwritepos;        // Write pointer into the output (overlap-add) buffer
int outreadpos;         // Read pointer into the output buffer
int inputBufferLength;  // Length of the input buffer (in samples)
int outputBufferLength; // Length of the output buffer (in samples)
int sampsincefft;       // Counter of how many samples have elapsed since last FFT
float *windowBuffer;    // Buffer that holds the (pre-calculated) window function
int windowBufferLength; // Length of the window function
float *fftTimeDomain;   // Buffer that holds time-domain samples for the FFT calculation
float *fftFrequencyDomain; // Buffer that holds frequency-domain samples from FFT
float fftScaleFactor;   // Scaling factor to normalize output level; depends on window/hop sizes

int fftTransformSize_;  // Size of the FFT calculation (in samples); normally equals
                        // window size but could be longer
int hopSize_;           // Hop size parameter (in samples)

// Collect the audio samples in the input buffer. When we've reached the next
// hop interval, calculate the FFT and process the pitch shift.
```

Robotisation VST II

```
for (int i = 0; i < numSamples; ++i)
{
    const float in = channelData[i];

    // Store the next buffered sample in the output. Do this first before anything
    // changes the output buffer-- we will have at least one FFT size worth of data
    // stored and ready to go. Set the result to 0 when finished in preparation for the
    // next overlap/add procedure.
    channelData[i] = outputBufferData[outreadpos];
    outputBufferData[outreadpos] = 0.0;
    if(++outreadpos >= outputBufferLength) outreadpos = 0;

    // Store the current sample in the input buffer, incrementing the write pointer. Also
    // increment how many samples we've stored since the last transform. If it reaches
    // the hop size, perform an FFT and any frequency-domain processing.
    inputBufferData[inwritepos] = in;
    if (++inwritepos >= inputBufferLength) inwritepos = 0;
```

Robotisation VST III

```
if (++sampsincefft >= hopSize_)
   {
       sampsincefft = 0;

       // Find the index of the starting sample in the buffer. When the buffer length
       // is equal to the transform size, this will be the current write position but
       // this code is more general for larger buffers.
       int inputBufferStartPosition = (inwritepos + inputBufferLength
                                        - fftTransformSize_) % inputBufferLength;

       // Window the buffer and copy it into the FFT input
       int inputBufferIndex = inputBufferStartPosition;
       for(int fftBufferIndex = 0; fftBufferIndex < fftTransformSize_; fftBufferIndex++)
       {
           // Set real part to windowed signal; imaginary part to 0.
           fftTimeDomain[fftBufferIndex][1] = 0.0;
           if(fftBufferIndex >= windowBufferLength) // Safety check, in case window
                                               // isn't ready
               fftTimeDomain[fftBufferIndex][0] = 0.0;
           else
               fftTimeDomain[fftBufferIndex][0] = windowBuffer[fftBufferIndex]
               * inputBufferData[inputBufferIndex];
           inputBufferIndex++;
           if(inputBufferIndex >= inputBufferLength)
               inputBufferIndex = 0;
       }
```

Robotisation VST IV

```
// Perform the FFT on the windowed data, going into the frequency domain.
       // Result will be in fftFrequencyDomain
       fftw_execute(fftForwardPlan_);

       // ********** PHASE VOCODER PROCESSING GOES HERE **************
       // This is the place where frequency-domain calculations are made
       // on the transformed signal. Put the result back into fftFrequencyDomain
       // before transforming back.
       // ************************************************************

       for(int bin = 0; bin < fftTransformSize_; bin++)
       {
           float amplitude = sqrt(fftFrequencyDomain[bin][0]*fftFrequencyDomain[bin][0]
                 + fftFrequencyDomain[bin][1]*fftFrequencyDomain[bin][1]);

           // Set the phase of each bin to 0. phase = 0 means the signal is entirely
           // positive-real, but the overall amplitude is the same as before.
           fftFrequencyDomain[bin][0] = amplitude;
           fftFrequencyDomain[bin][1] = 0.0;
       }

       // Perform the inverse FFT to get back to the time domain. Result wll be
       // in fftTimeDomain. If we've done it right (kept the frequency domain
       // symmetric), the time domain resuld should be strictly real allowing us
       // to ignore the imaginary part.
       fftw_execute(fftBackwardPlan_);
```

Robotisation VST V

```
        // Add result to output buffer, starting at current write position
        // (Output buffer will have been zeroed after reading the last time around)
        // Output needs to be scaled by the transform size to get back to original
        // amplitude: this is a property of how fftw is implemented. Scaling will also
        // need to be adjusted based on hop size to get the same output level (smaller
        // hop size produces more overlap and hence higher signal level)
        int outputBufferIndex = outwritepos;
        for(int fftBufferIndex = 0; fftBufferIndex < fftTransformSize_; fftBufferIndex++)
        {
            outputBufferData[outputBufferIndex] += fftTimeDomain[fftBufferIndex][0] * fftScaleFactor;
            if(++outputBufferIndex >= outputBufferLength) outputBufferIndex = 0;
        }

        // Advance the write position within the buffer by the hop size
        outwritepos = (outwritepos + hopSize_) % outputBufferLength;
    }
}
```

Robotisation

- Window size affects performance
  - Use short window to capture just one period of waveform per segment

Whisperisation

- Whisperisation = erasing pitch cues
  - Implemented by randomising DFT phases before reconstruction
  - Similar to Robotization
  - Short window lengths work best

Whisperisation VST

- adapt previous robotization code example to perform whisperization
  - basic overlap-add structure stays
  - only need to change lines following FFT calculation

```
int for(int bin = 0; bin <= fftTransformSize_ / 2; bin++)
{
    float amplitude = sqrt(fftFrequencyDomain[bin][0]*fftFrequencyDomain[bin][0] +
                           fftFrequencyDomain[bin][1]*fftFrequencyDomain[bin][1]);

    // This is what we would use to exactly reconstruct the signal:
    // float phase = atan2(fftFrequencyDomain[bin][1], fftFrequencyDomain[bin][0]);
    // Instead, use this to scramble phase:
    float phase = 2.0 * M_PI * (float)rand() / (float)RAND_MAX;

    // Set phase of each bin to 0. phase = 0 means signal entirely positive-real,
    // but overall amplitude same as before.
    fftFrequencyDomain[bin][0] = amplitude * cos(phase);
    fftFrequencyDomain[bin][1] = amplitude * sin(phase);

    // FFTs of real signals are conjugate-symmetric. We need to maintain that symmetry
    // to produce a real output, even as we randomize the phase.
    if(bin > 0 && bin < fftTransformSize_ / 2) {
        fftFrequencyDomain[fftTransformSize_ - bin][0] = amplitude * cos(phase);
        fftFrequencyDomain[fftTransformSize_ - bin][1] = -amplitude * sin(phase);
    }
}
```

Time scaling and pitch shifting

- Analysis identical to any standard phase vocoder
- Synthesis hop size varied for time compression/expansion
  - Changes time scale without changing pitch
- If pitch shifting, interpolate back to original time scale
  - Now pitch changed, but time scale is unchanged

Pitch shifting VST I

```
int outwritepos;          // Temporary write pointer
float *inputBufferData;   // Buffered input samples awaiting FFT
int inputBufferLength;    // Length of the input buffer (in samples)
float *outputBufferData;  // Buffered output samples for overlap-add
int outputBufferLength;   // Length of the output buffer (in samples)
int sampsincefft;         // Counter of how many samples have elapsed since last FFT
float *windowBuffer;      // Buffer that holds the analysis window function
int windowBufferLength;   // Length of the analysis window function
float *synthWindowBuffer; // Buffer that holds the synthesis window function
int synthesisWindowLength;// Length of the synthesis window
float *fftTimeDomain;   // Buffer that holds time-domain samples for the FFT calculation
float *fftFrequencyDomain; // Buffer that holds frequency-domain samples from FFT
float fftScaleFactor;   // Scaling factor to normalize output level; depends on window/hop sizes
float *resampledOutput; // Buffer holding resampled (interpolated) output from FFT
float **lastPhase;      // Previous phase values for each bin and channel
float **psi;            // Adjusted phase values for each bin and channel

int fftTransformSize_;  // Size of the FFT calculation (in samples); normally equals
                        // window size but could be longer
double pitchRatio_;     // Ratio of output to input frequency
int analysisHopSize_;   // Hop size parameter for input (in samples)
int synthesisHopSize_;  // Hop size parameter for output (in samples)
                        // synthesis / analysis size should match pitchRatio_
```

Pitch shifting VST II

- executes the pitch shift for one (overlapped) FFT window

```
fftw_execute(fftForwardPlan_);
for (int i = 0; i < fftTransformSize_; i++) {
  // Convert bin into magnitude-phase representation
  double magnitude = sqrt(fftFrequencyDomain[i][0] * fftFrequencyDomain[i][0]
                         +fftFrequencyDomain[i][1] * fftFrequencyDomain[i][1]);
  double phase = atan2(fftFrequencyDomain[i][1], fftFrequencyDomain[i][0]);

  // Calculate frequency for this bin
  double frequency = 2.0 * M_PI * (double)i / fftTransformSize_;

  // Increment the phase based on frequency and hop sizes
  double deltaPhi = (frequency * analysisHopSize_) +
                    princArg(phase - lastPhase[i][channel] - (frequency * analysisHopSize_));
  lastPhase[i][channel] = phase;
  psi[i][channel] = princArg(psi[i][channel] + deltaPhi * synthesisHopSize_);

  // Convert back to real-imaginary form
  fftFrequencyDomain[i][0] = magnitude * cos(psi[i][channel]);
  fftFrequencyDomain[i][1] = magnitude * sin(psi[i][channel]);
}

fftw_execute(fftBackwardPlan_); // Perform inverse FFT
```

Pitch Shifting VST III

```
// Resample output using linear interpolation to stretch it
double outputLength = floor(fftTransformSize_ / pitchRatio_);

for(int i = 0; i < outputLength; i++) {
    x = i * fftTransformSize_ / outputLength;
    ix = floor(x);
    dx = x - (double)ix;
    resampleOutput[i] = fftTimeDomain[ix]*(1.0 - dx) + fftTimeDomain[(ix+1)%fftTransformSize_]*dx;
}

// Add the result to the output buffer, starting at the current write position
int outputBufferIndex = outwritepos;

for(int fftBufferIndex = 0; fftBufferIndex < outputLength; fftBufferIndex++) {
    if (fftBufferIndex > synthesisWindowLength) outputBufferData[outputBufferIndex] += 0;
    else outputBufferData[outputBufferIndex] += resampleOutput[fftBufferIndex] * fftScaleFactor *
                                                synthesisWindowBuffer[fftBufferIndex];
    if(++outputBufferIndex >= outputBufferLength) outputBufferIndex = 0;
}

// Advance the write position within the buffer by the hop size
// (Use original hop size since we have resampled output back to expected length)
outwritepos = (outwritepos + analysisHopSize_) % outputBufferLength_;
```