

Perception & Multimedia Computing

Week 18 – Finishing Signals & Systems; FFT & Effects

Michael Zbyszyński

Lecturer, Department of Computing
Goldsmiths University of London

Last time...

The convolution process

Filter equations

Using the *frequency response*
to design and reason about
effects

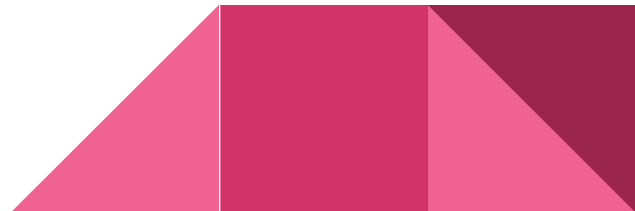
Filter design tools

Exercise #1

$$h = [0.5, 1.0]$$

$$x = [1.0, 0.5]$$

What is the output of H when x is input?



Exercise #1

Answer: [0.5, 1.25, 0.5]

Solve by convolving x and h ,

or by realising that filter equation is

$$y[n] = 0.5 * x[n] + 1.0 * x[n-1]$$


Exercise #2

$$h = [-1.0, 1.0]$$

$$x = [0.5, 1.0]$$

What is the output of H when x is input?



Exercise #2

Answer: [-1, 0.5, 0.5]

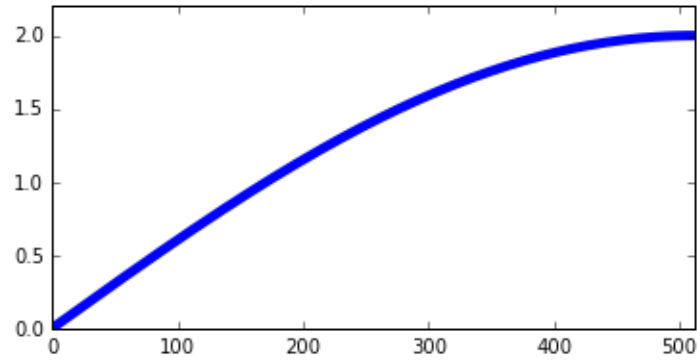
Solve by convolving x and h ,

or by realising that filter equation is

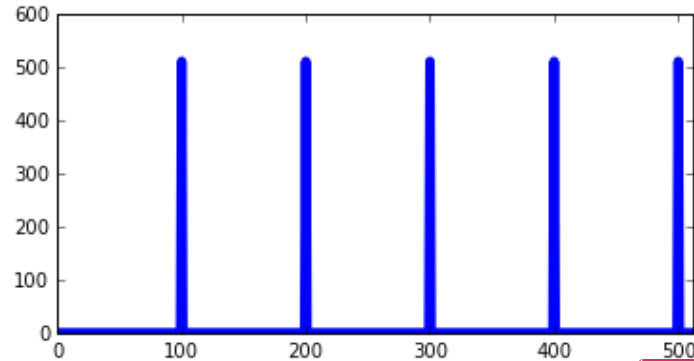
$$y[n] = -1 * x[n] + 1 * x[n-1]$$


Exercise #3

Frequency response:

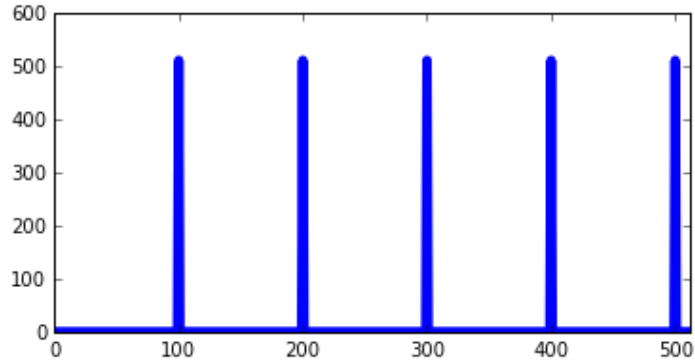
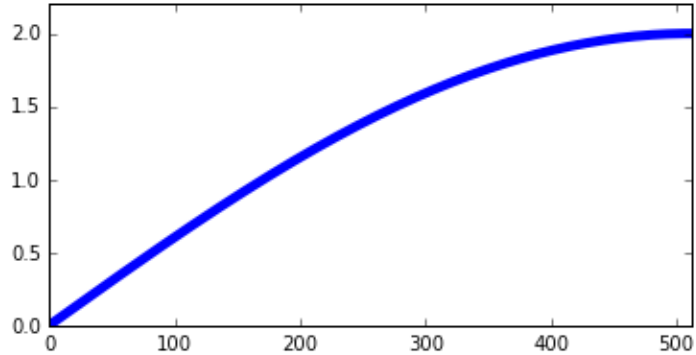


Spectrum of input:

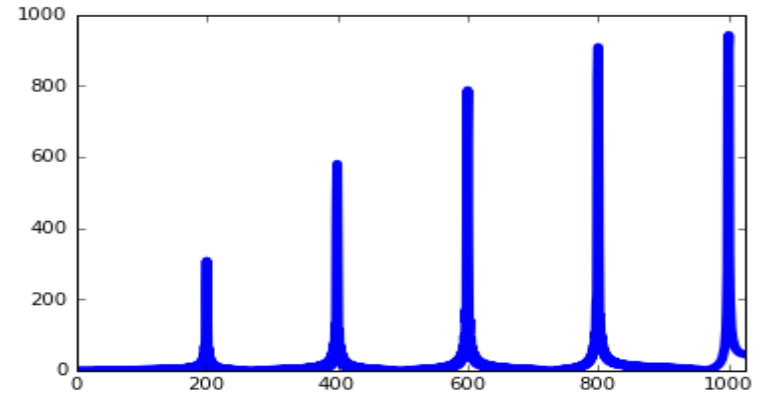


What is spectrum of output?

Exercise #3

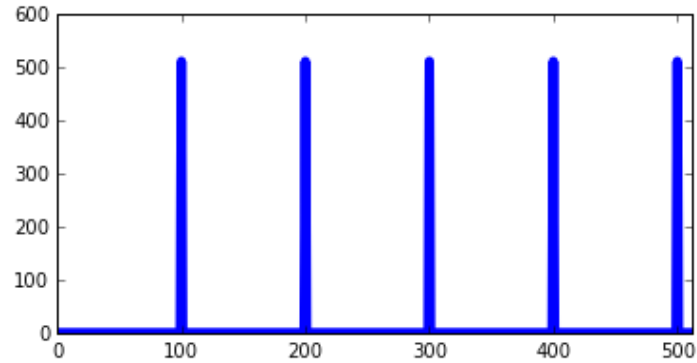


Spectrum of output:
Multiply bin by bin:

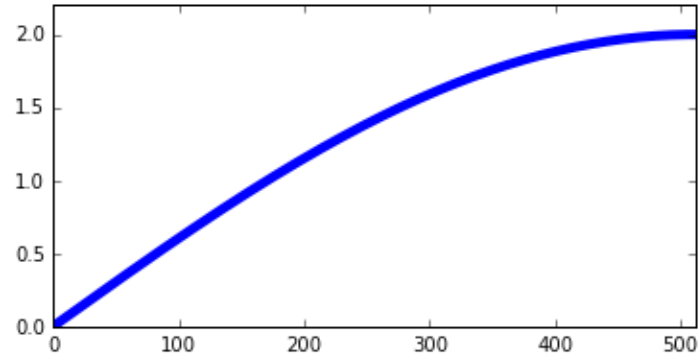


Exercise #4

Frequency response:



Spectrum of input:



What is spectrum of output?

Exercise #4

Answer: It's the same as the previous exercise.



Today

Finishing up filters:
feedback!

Multiplication &
convolution

Revisiting FFT

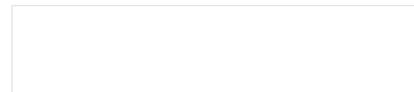
More effects...

MIND

A Learning Secret: Don't Take Notes with a Laptop

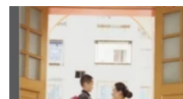
Students who used longhand remembered more and had a deeper understanding of the material

By Cindi May on June 3, 2014  26 [Véalo en español](#)



ADVERTISEMENT | REPORT AD

READ THIS NEXT

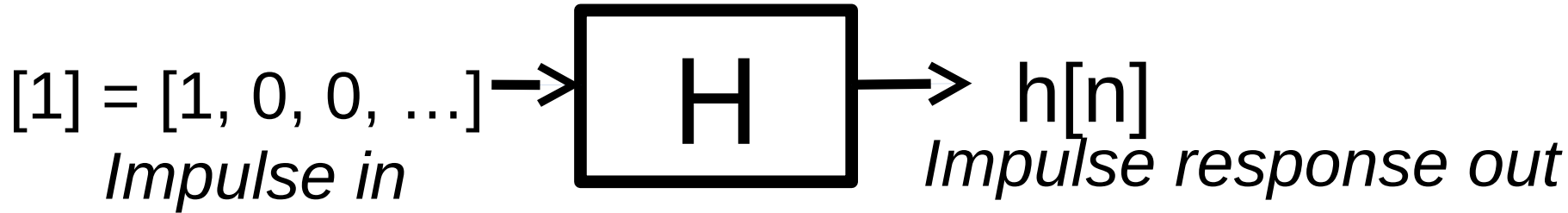


The Science of Education:
Back to School

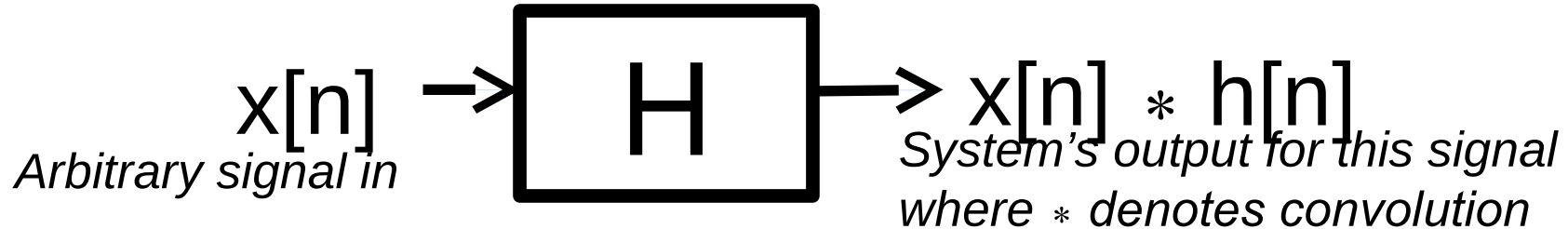
<https://www.scientificamerican.com/article/a-learning-secret-don-t-take-notes-with-a-laptop/>

Filter review

For any linear, time-invariant system H , if we know the system's impulse response:



Then we can compute the output of the system for any new input, by convolving that input with the impulse response.



What is a filter equation?


Express $y[n]$ as a weighted sum of $x[n], x[n-1], x[n-2], \text{etc.}$

the current input value → $x[n]$

the input value 1 sample ago (i.e., the previous input) → $x[n-1]$

the input value 2 samples ago → $x[n-2]$

← the current output $y[n]$



Generic form

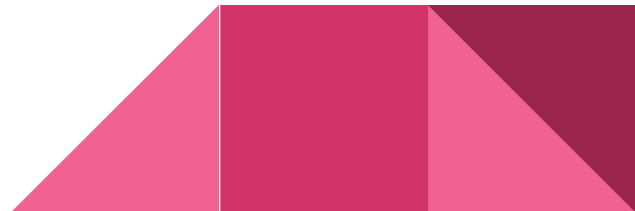
$$y[n] = b_0 * x[n] + b_1 * x[n-1] \\ + b_2 * x[n-2] + \dots + b_Q * x[n-Q]$$

b_0 , b_1 , etc are called *filter coefficients*

Example:

$$y[n] = 0.5 * x[n] + 1 * x[n-1]$$

What is output when $x = [1, 2]$?



Impulse response & filter equation

$$h[n] = [b_0, b_1, b_2, \dots, b_Q]$$

$$y[n] = b_0 * x[n] + b_1 * x[n-1] \\ + b_2 * x[n-2] + \dots + b_Q * x[n-Q]$$

Check by computing y when x is unit impulse[1]




A VERY useful property

If $y[n] = x[n] * h[n]$, then $Y_k = X_k \cdot H_k$

In English:

You can compute the output of a system H by convolving the input with the impulse response of the system, h .

You can equivalently compute the spectrum of the output by multiplying the spectrum of the input, bin by bin, with the spectrum of the impulse response (which is the frequency response).



A VERY useful property

Colloquially:

Convolution in the time domain is equivalent to multiplication in the frequency domain.

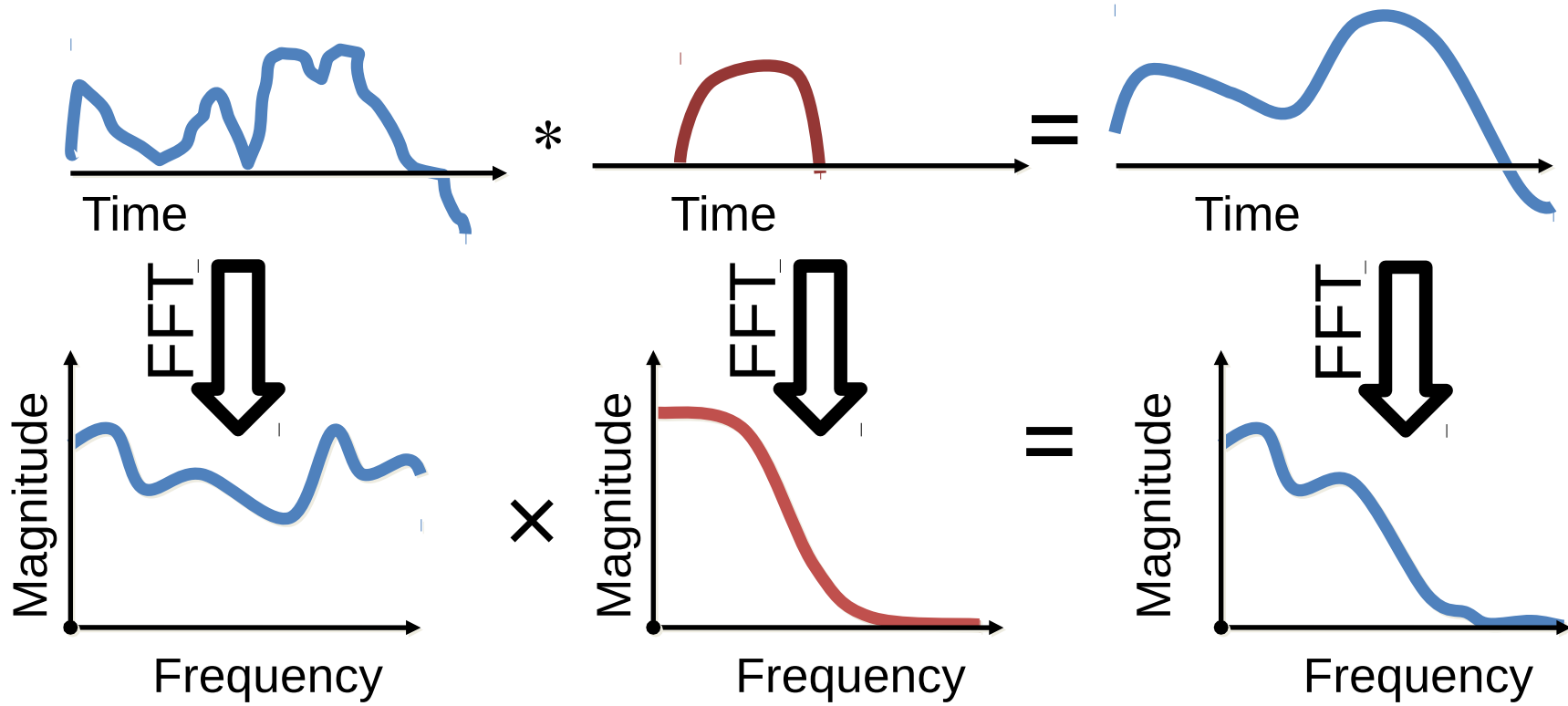
Also: Multiplication in the time domain is equivalent to convolution in the frequency domain.

(more on this later)



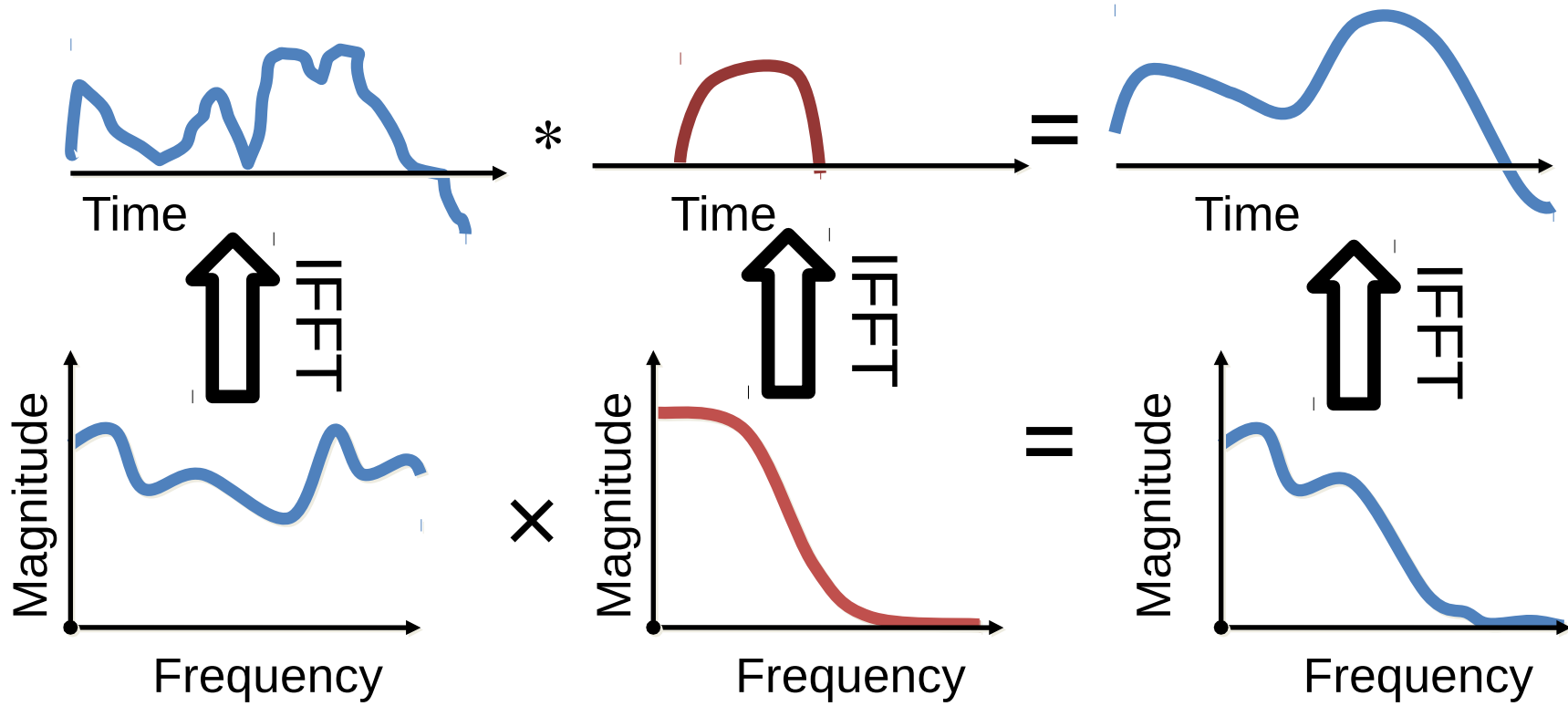
Convolution & Multiplication

Reasoning about system effects on an input:



Convolution & Multiplication

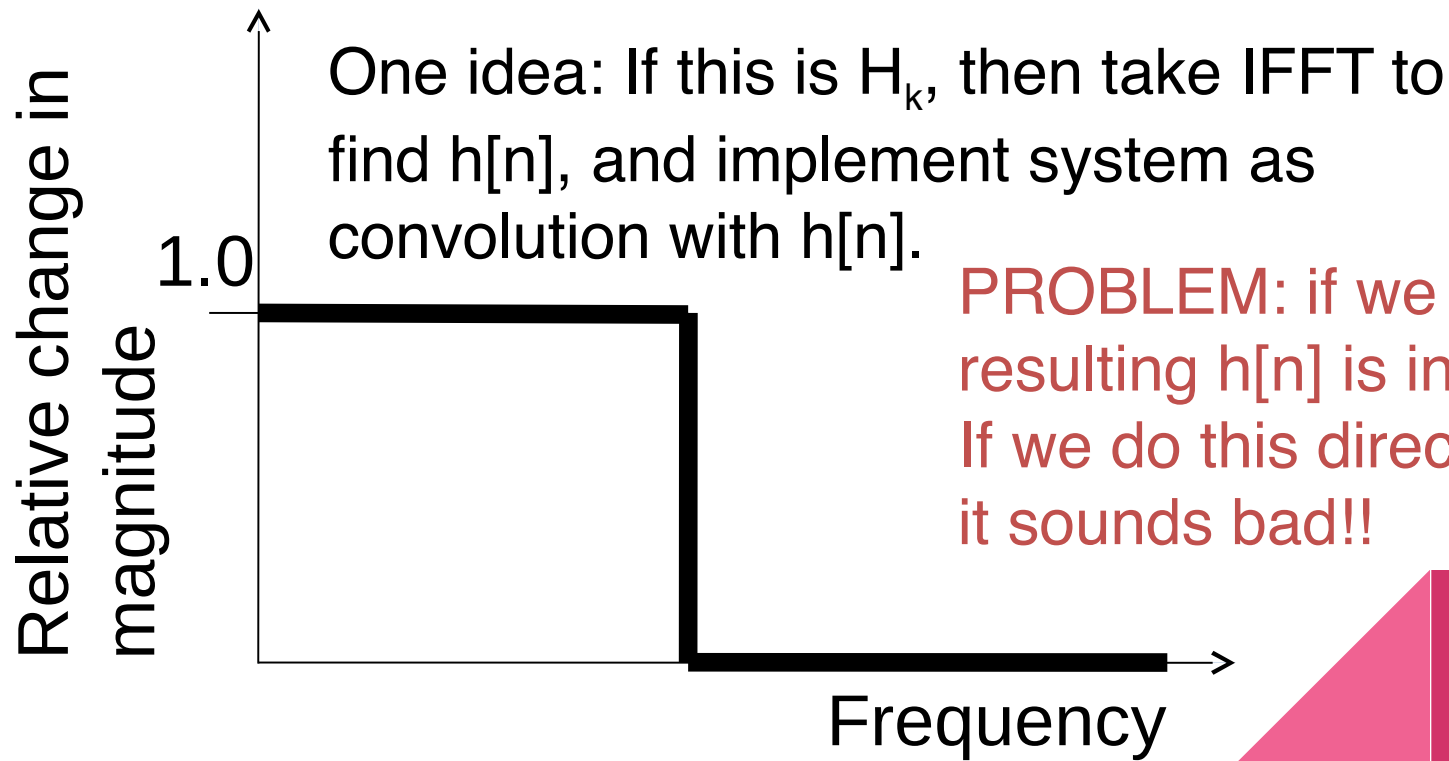
Reasoning about system effects on an input:



How does this help us?

1. Frequency response tells us how each frequency in the input will be changed by the filter/effect.
2. We can design filters by creating the desired frequency response, then computing the impulse response / filter equation.

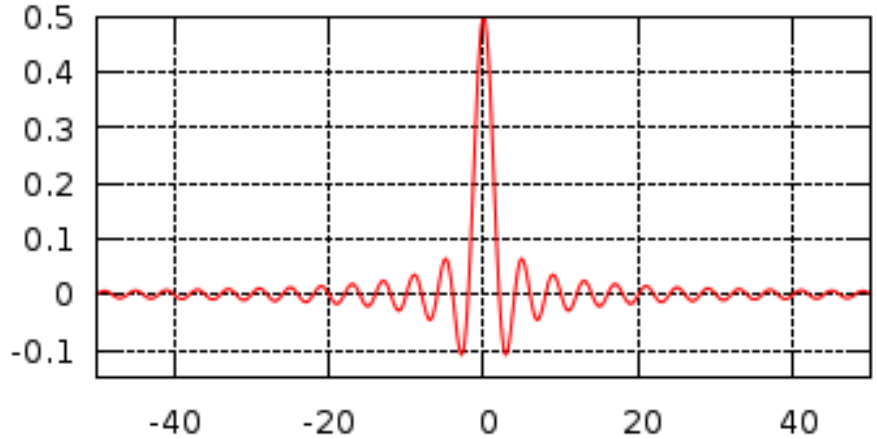
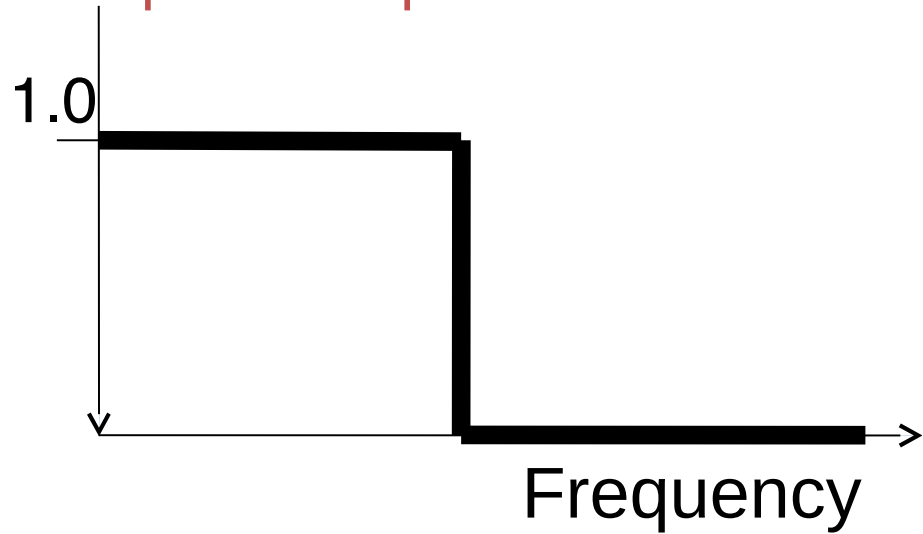
Constructing a low-pass filter



PROBLEM: if we do this, then resulting $h[n]$ is infinitely long! If we do this directly with FFT, it sounds bad!!

Big problems

The sharper the change in frequency, the longer the impulse response.



impulse response



Feedback filters

What if we make $y[n]$ dependent on previous *output* values as well?

$$\begin{aligned} y[n] = & b_0 * x[n] + b_1 * x[n-1] \\ & + b_2 * x[n-2] + \dots + b_Q * x[n-Q] \\ & + a_1 * y[n-1] + a_2 * y[n-2] \\ & + \dots + a_M * y[n-M] \end{aligned}$$

This is an *infinite impulse response (IIR)* filter,
also called a feedforward filter.



Exercise

$$y[n] = x[n] + 0.5 * y[n-1] + 0.6 * y[n-2]$$

What is the output for $x = [1.0, 1.0]$?

Can use lfilter function in python:

```
In [90]: a = [1, -0.5, -0.6]
         b = [1]
         x = [1.0, 1.0, 0.0, 0.0, 0.0, 0.0]
         y = signal.lfilter(b, a, x)
         y
```

```
Out[90]: array([ 1.        ,  1.5       ,  1.35      ,  1.575     ,  1.5975    ,  1.74375])
```

Tradeoffs: IIR vs FIR

IIR:

- More computationally efficient: sharper frequency response with fewer coefficients
 - e.g., biquad filter: 3 feedforward & 2 feedback coefficients
- May do weird things to phase alignment
- Can “blow up” if you don’t design them right

FIR:

- More computationally expensive: need more coefficients (also introduces more delay)
- Can be “linear phase” same relative phase shift for all frequencies

Resources

Python: `scipy.signal.lfilter`, `scipy.signal.iirfilter`

Biquad filter calculator:

<http://www.earlevel.com/main/2013/10/13/biquad-calculator-v2/>

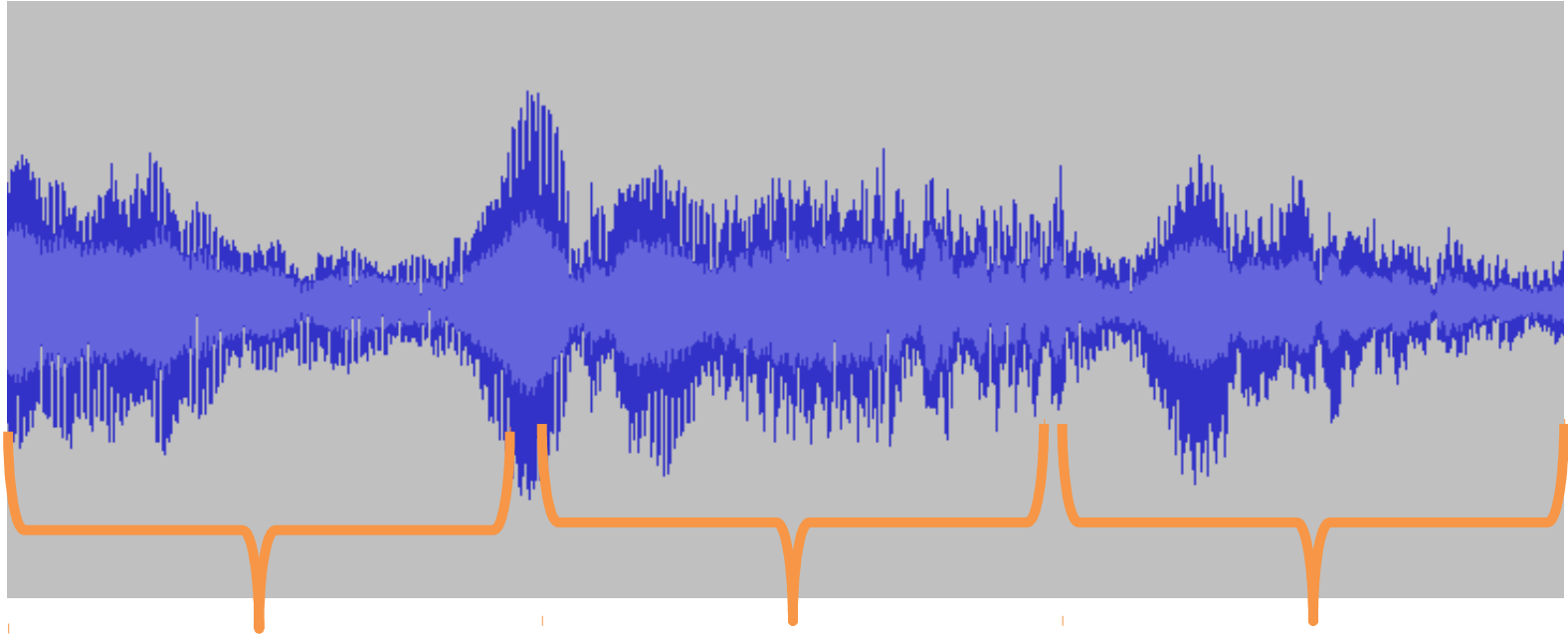
Longer discussion of IIR vs FIR:

<https://www.minidsp.com/applications/dsp-basics/fir-vs-iir-filtering>



Revisiting the FFT

Short-time Fourier Transform (“STFT”):
Analyze how a signal’s spectrum changes over time

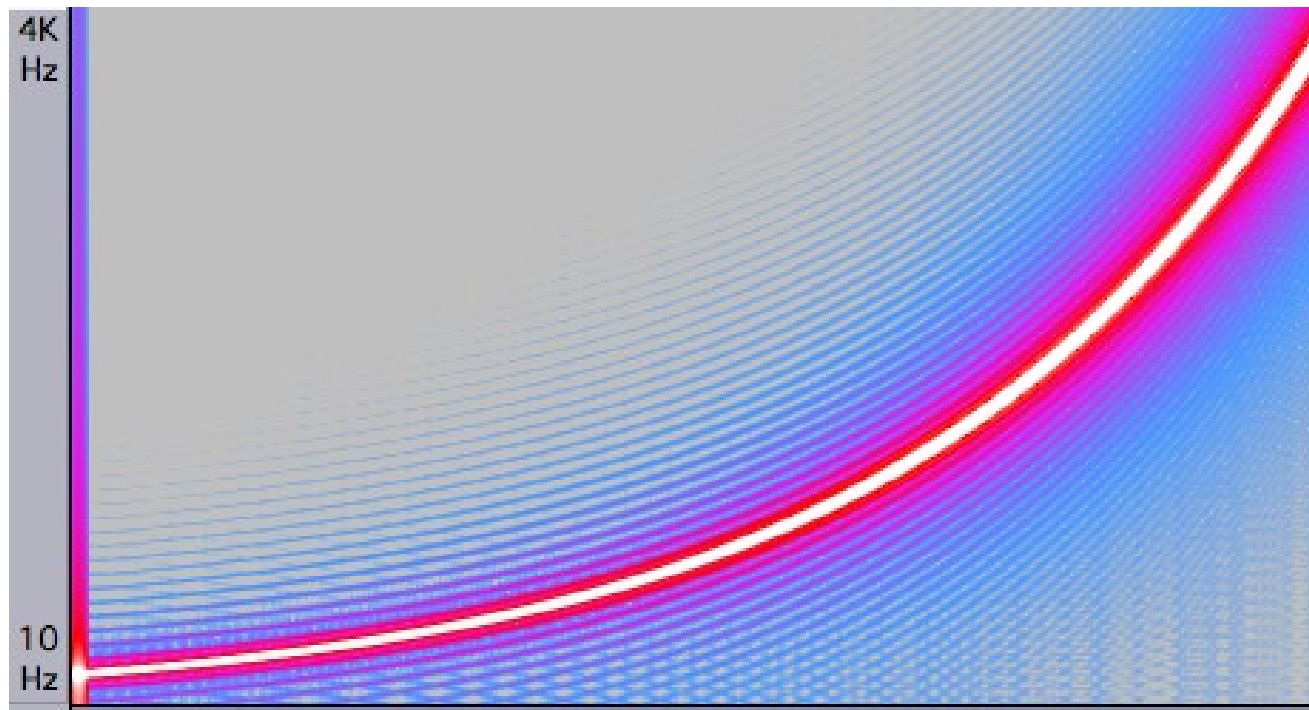


N-point FFT

N-point FFT

N-point FFT

What will you hear?



How many bins to use? (What should N be?)

More bins?

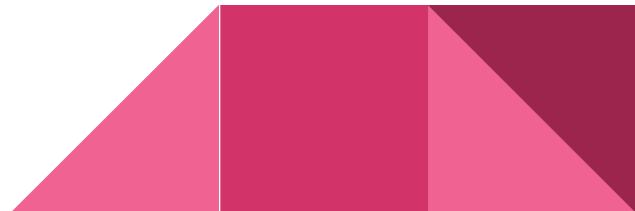
- Better frequency resolution

- Worse time resolution (FFT can't detect changes within the analysis frame)

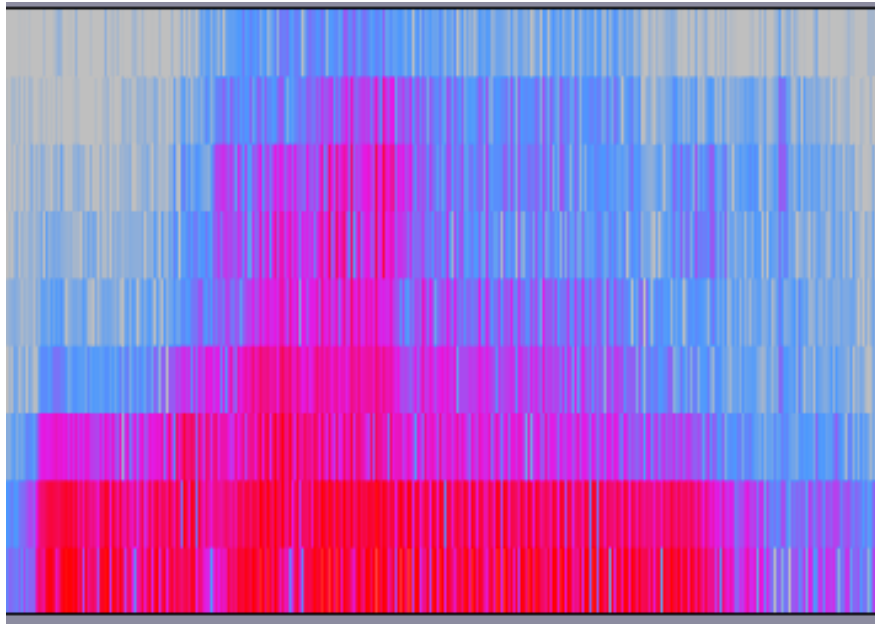
Fewer bins?

- Worse frequency resolution

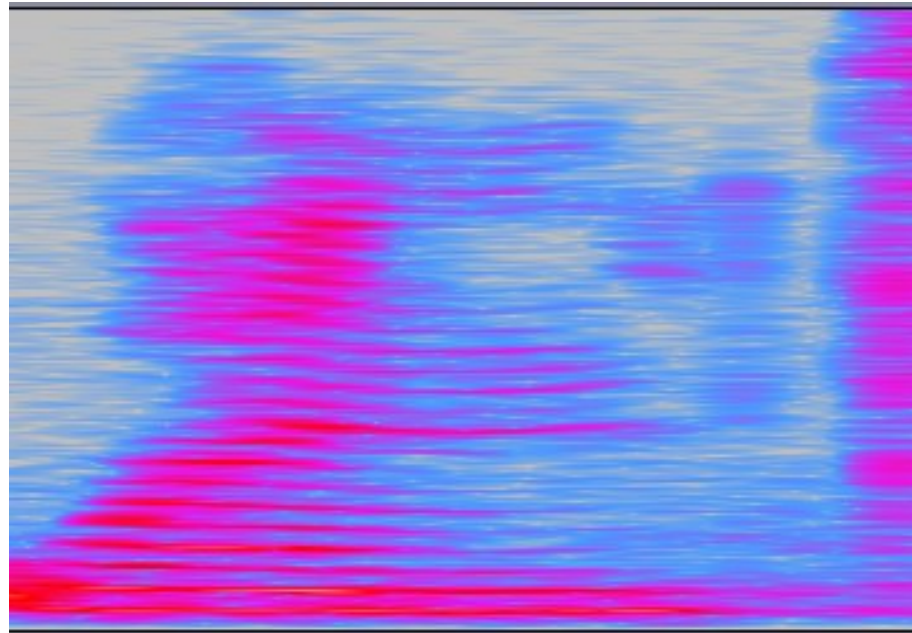
- Better time resolution



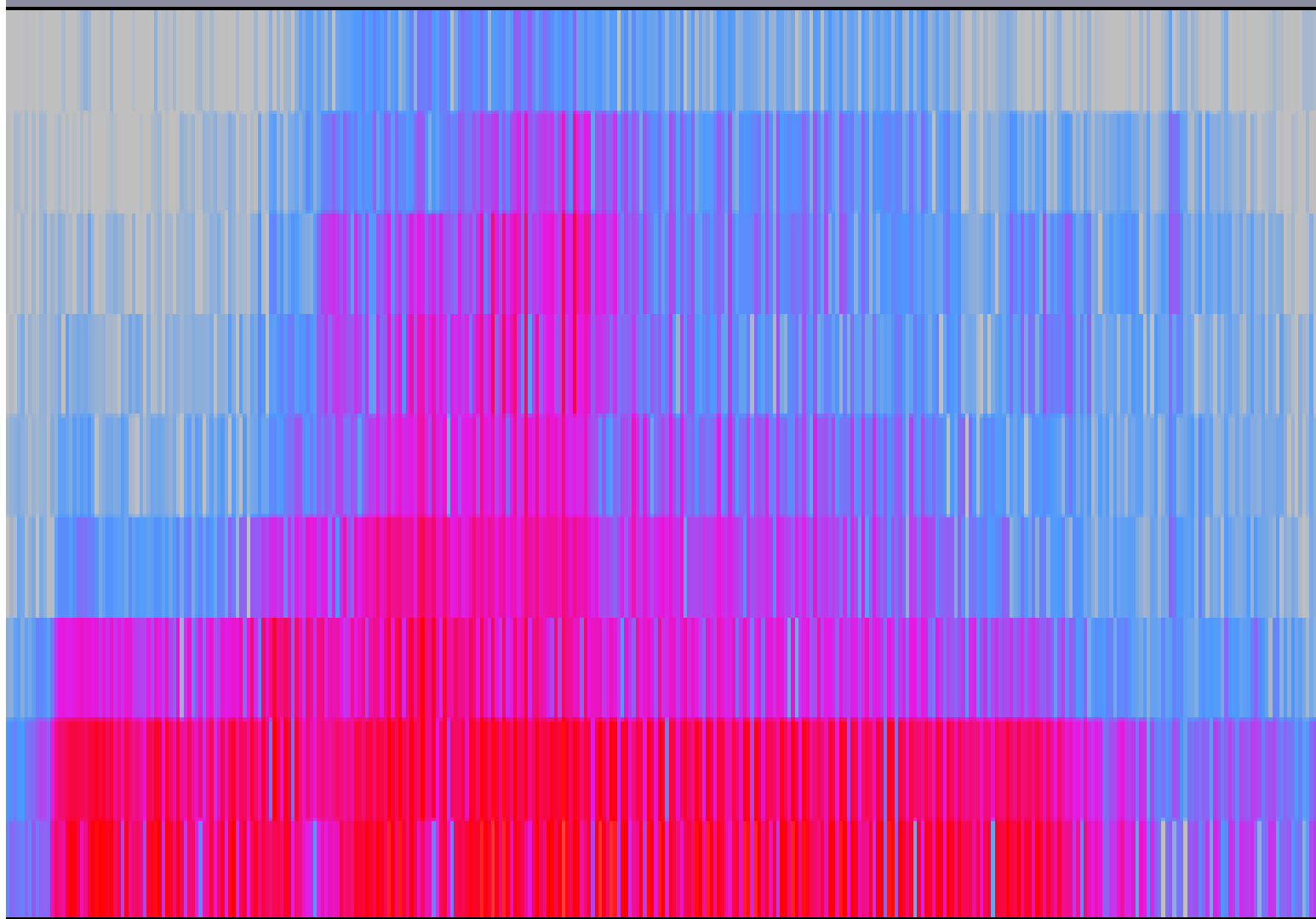
Time/Frequency tradeoff

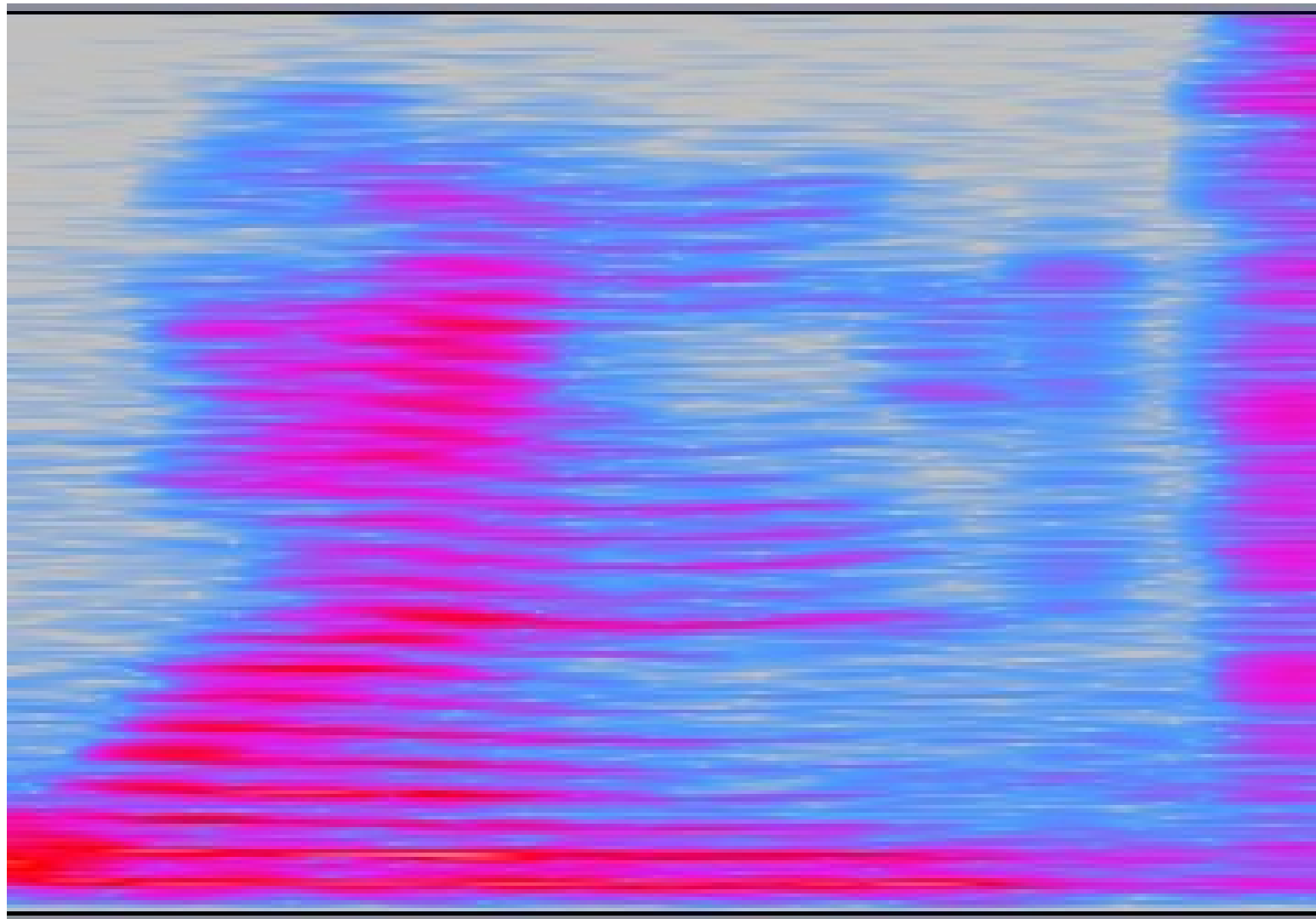


$N=64$

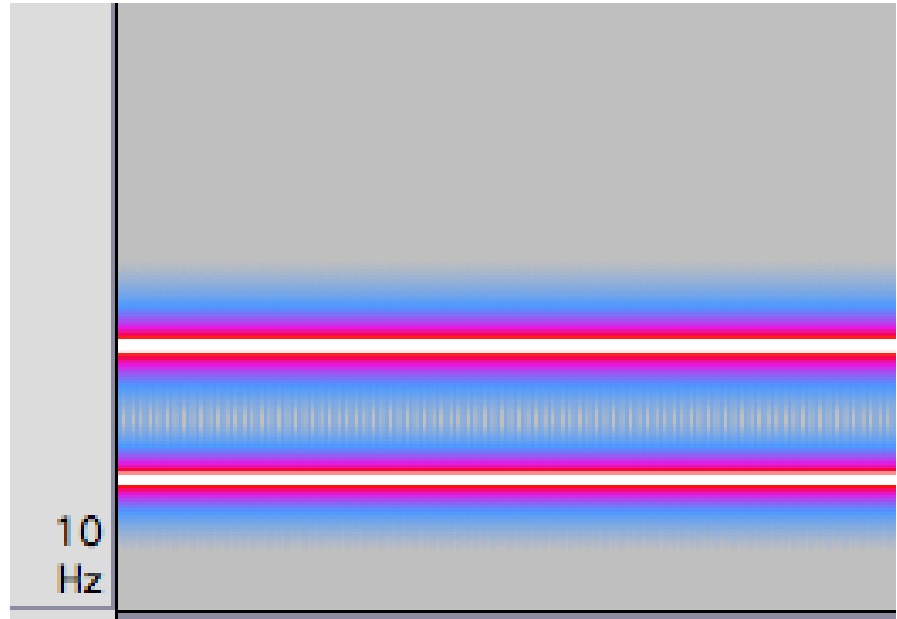
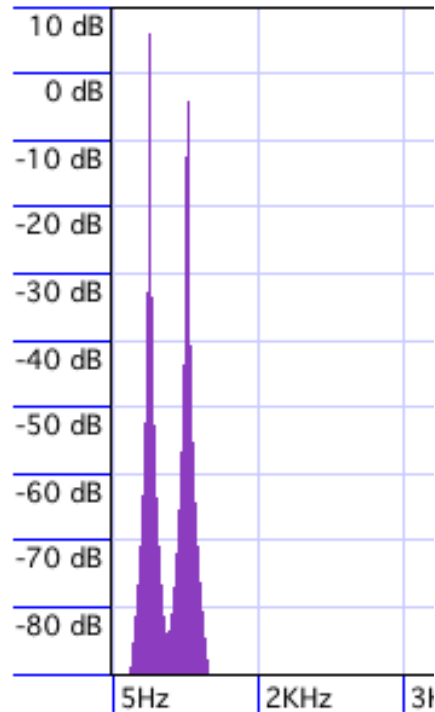


$N=4096$

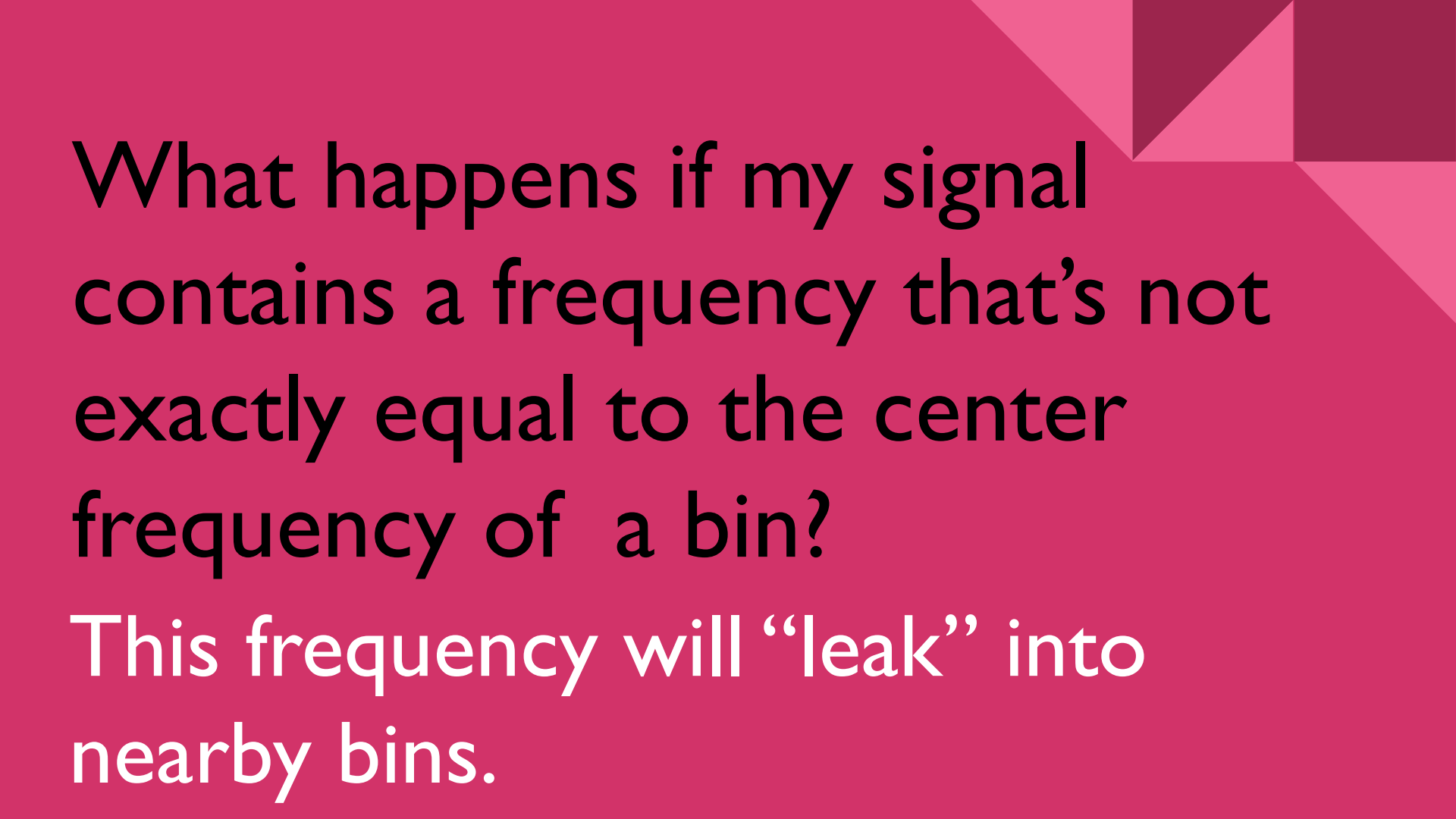




What's all that extra stuff in the spectrum?



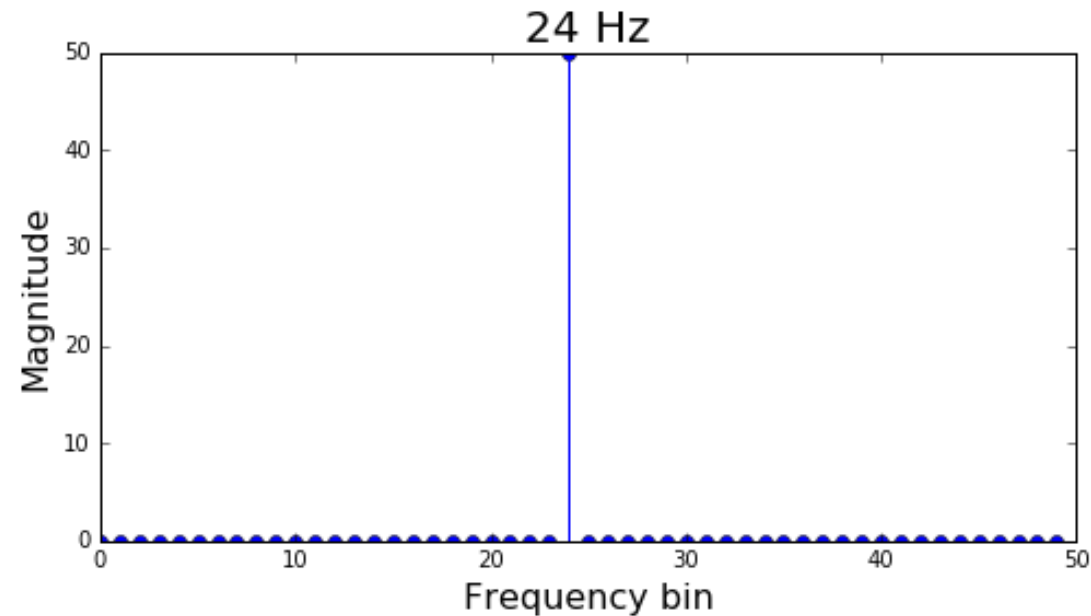
Not just clean peaks at frequencies and 0 elsewhere...



What happens if my signal
contains a frequency that's not
exactly equal to the center
frequency of a bin?

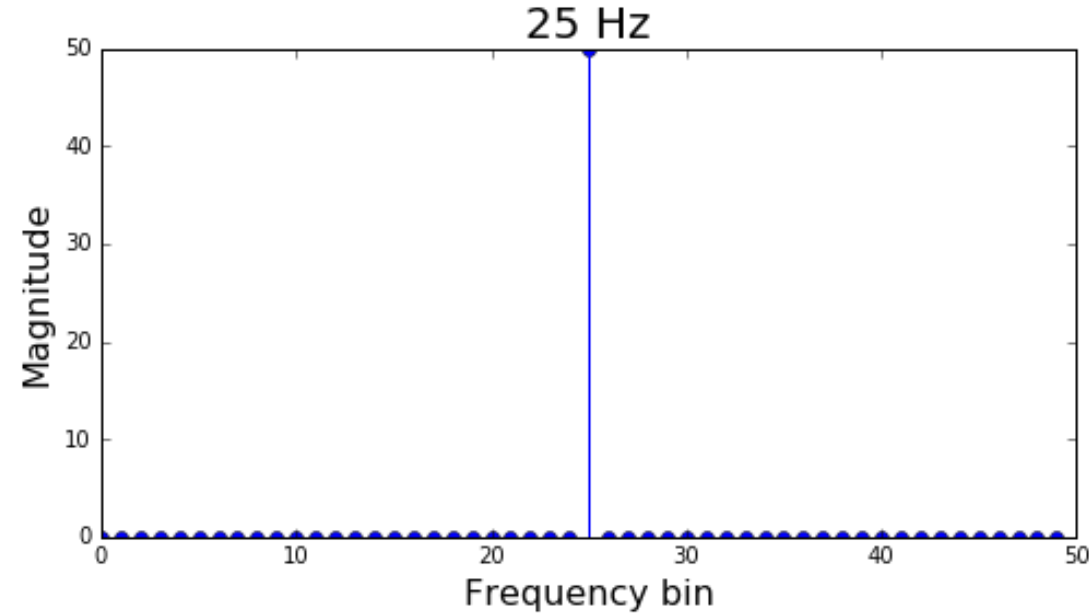
This frequency will “leak” into
nearby bins.

SR = 100Hz, sine at 24 Hz



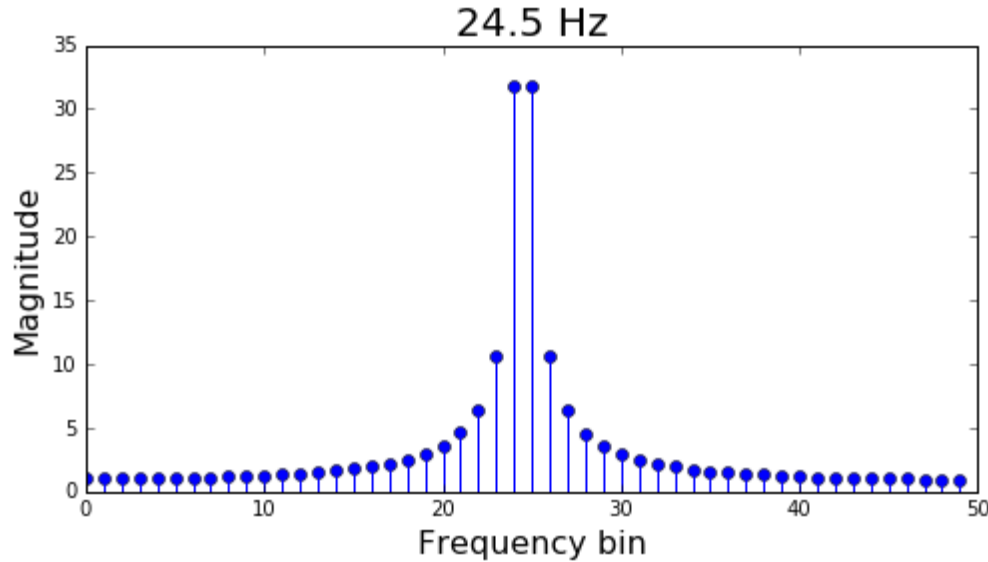
```
t = np.arange(0, 1, 1/100)
s1 = sin(2*pi*24*t)
stem(abs(fft.fft(s1))[0:50])
```

SR = 100Hz, sine at 25 Hz



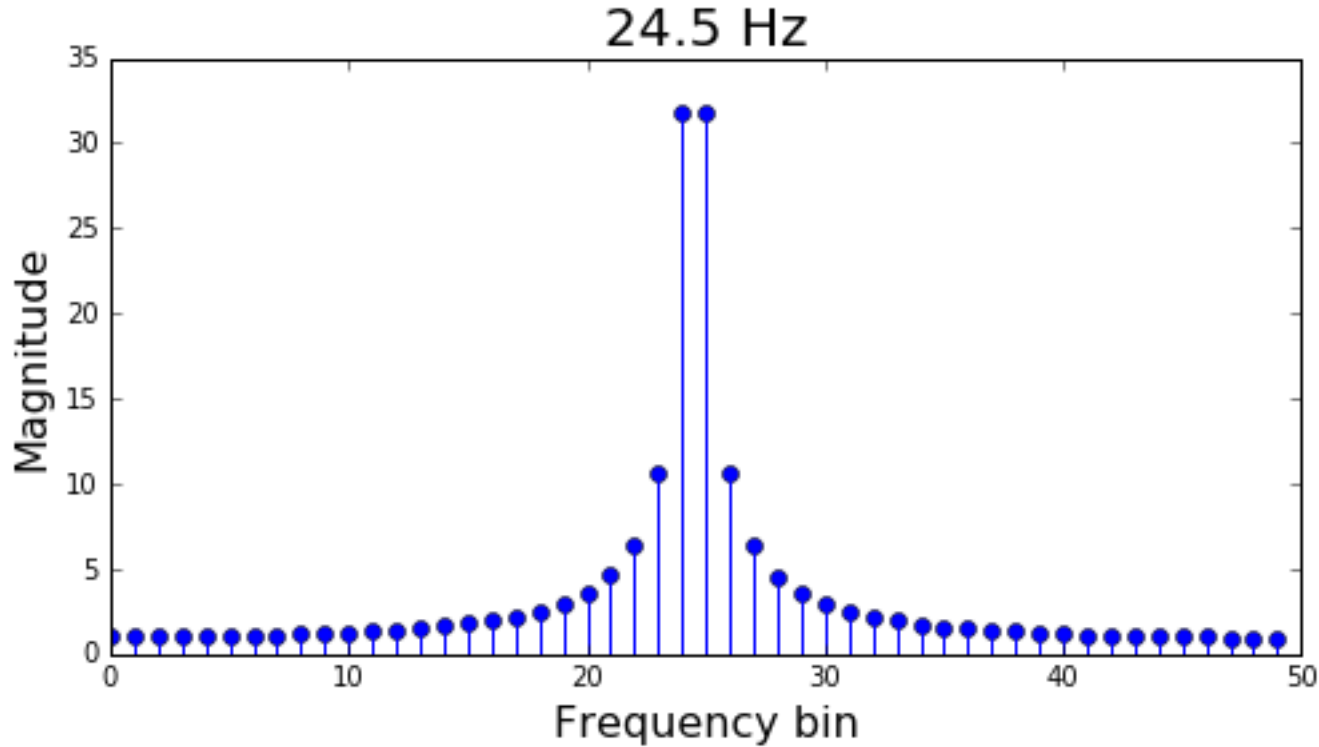
```
s2 = sin(2*pi*25*t)  
stem(abs(fft.fft(s2))[0:50])
```


SR = 100Hz, sine at 24.5 Hz

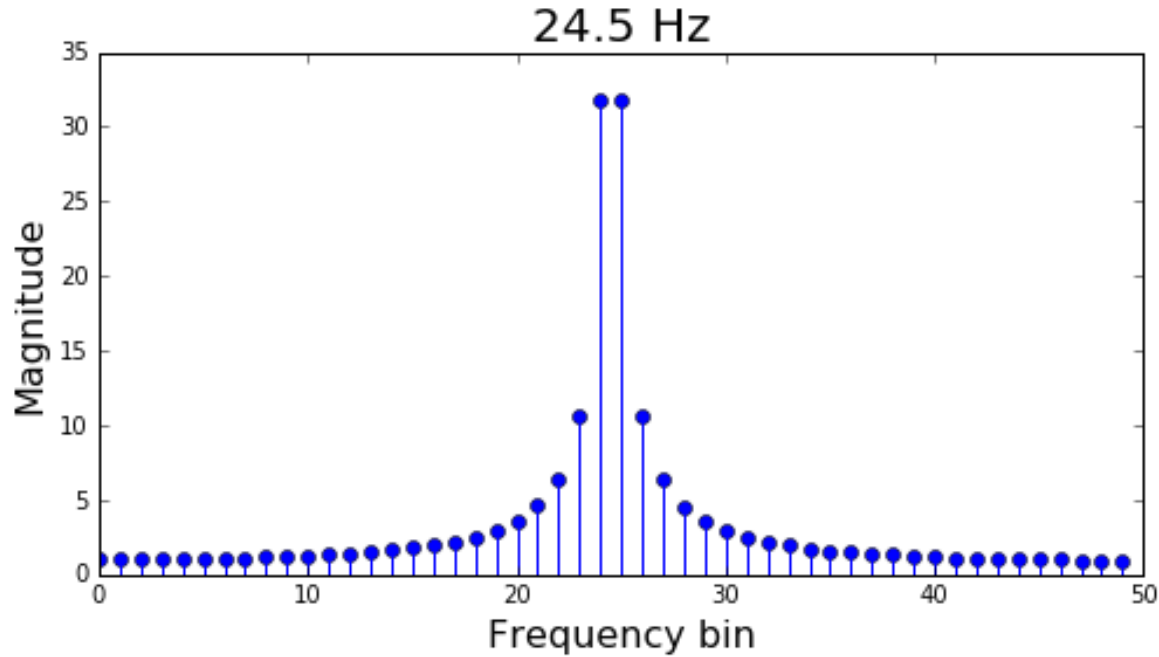


```
t = np.arange(0, 1, 1/100)
s1 = sin(2*pi*25*t)
s2 = sin(2*pi*26*t)
s3 = sin(2*pi*25.5*t)
stem(abs(fft.fft(s3))[0:50])
```

leakage!!

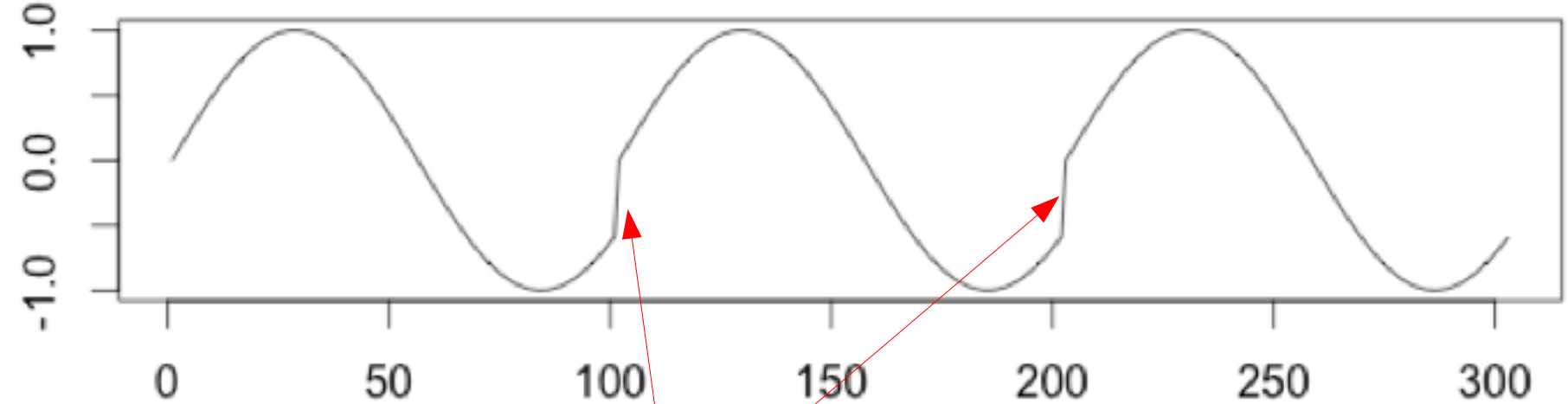


FFT answers the question, how can I combine sinusoids with the **specific** frequencies 0 , $(1/N)*SR$, $(2/N)*SR$, ... $SR/2$ to re-create my signal?



FFT answers the question, how can I combine sinusoids with the specific frequencies 0 , $(1/N)*SR$, $(2/N)*SR$, ... $SR/2$ to re-create my signal an infinitely repeating signal, where one repetition looks like my analysis frame?

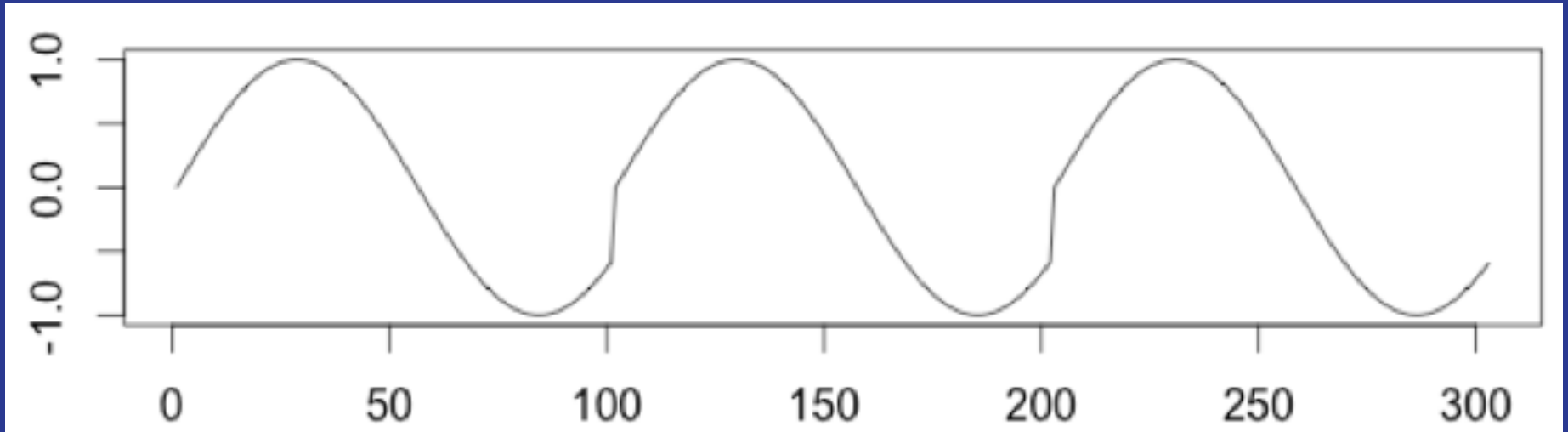
FFT treats your analysis frame as one period of an infinite, periodic signal.



“periodic” signal may have discontinuities

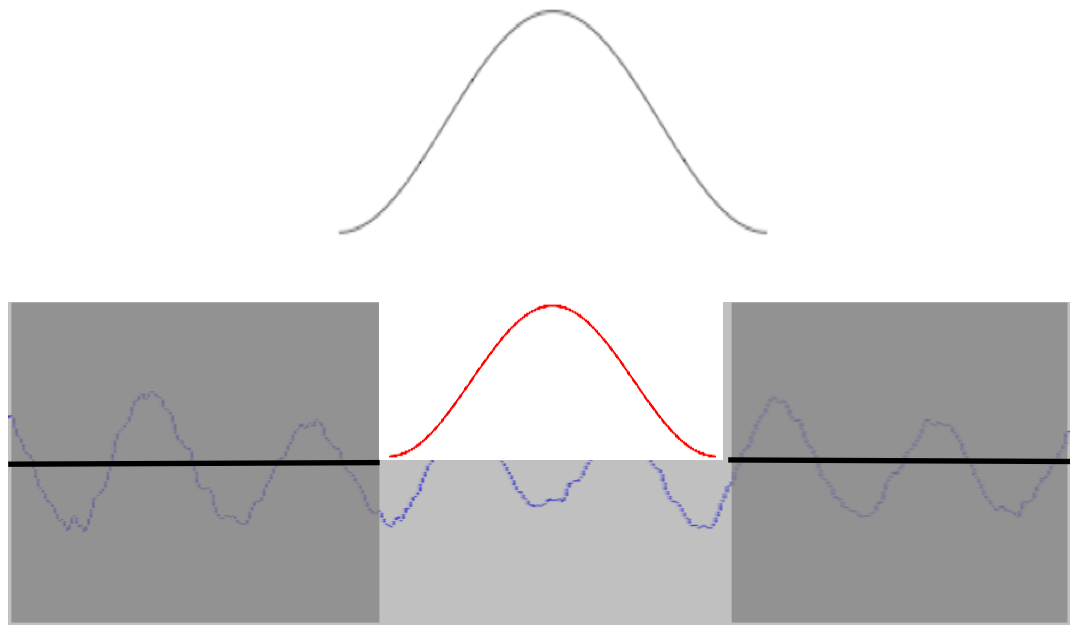
→ only representable with high frequency content

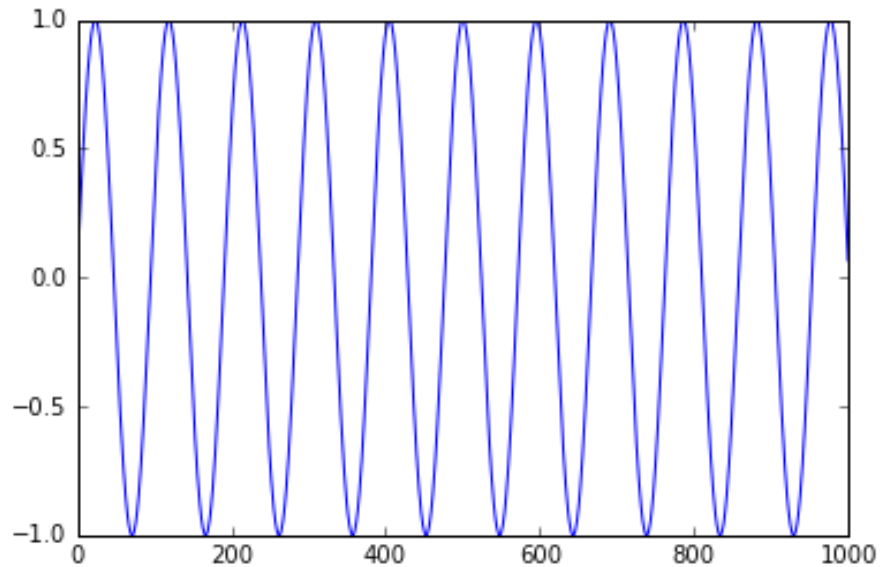
Windowing is a technique for getting rid of these discontinuities



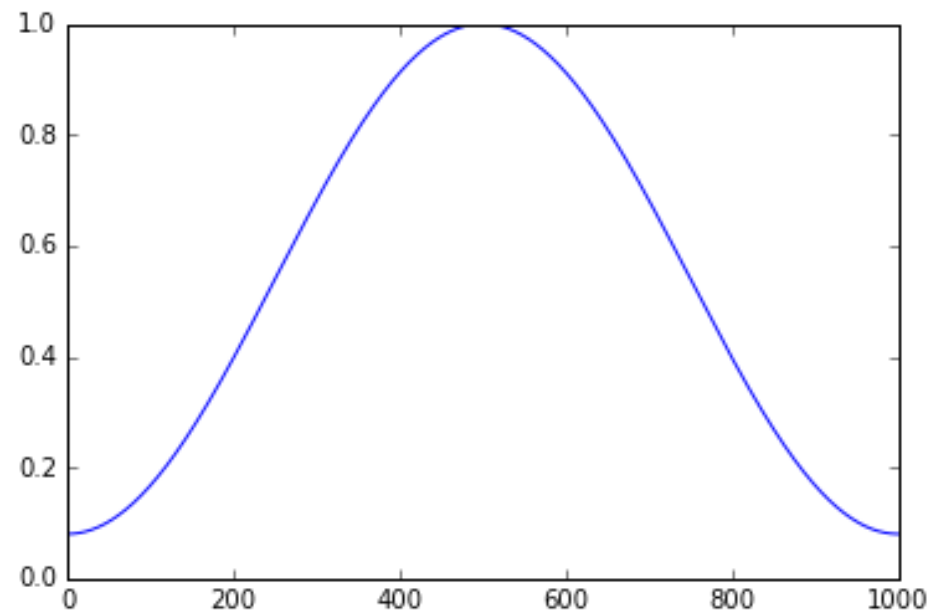
Windowing

Before taking FFT, multiply the signal with a smooth window that will get rid of sharp edges at either end of analysis frame



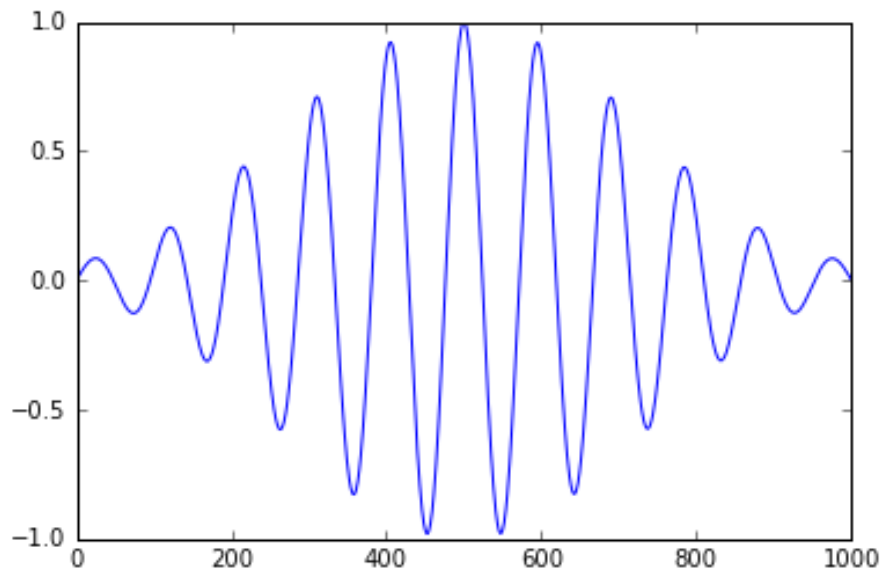


```
t = np.arange(0, 1, 1/1000)
s = sin(2*pi*10.5*t)
plot(s)
```



```
t = np.arange(0, 1, 1/1000)
s = sin(2*pi*10.5*t)
w = np.hamming(1000)
plot(w)
```

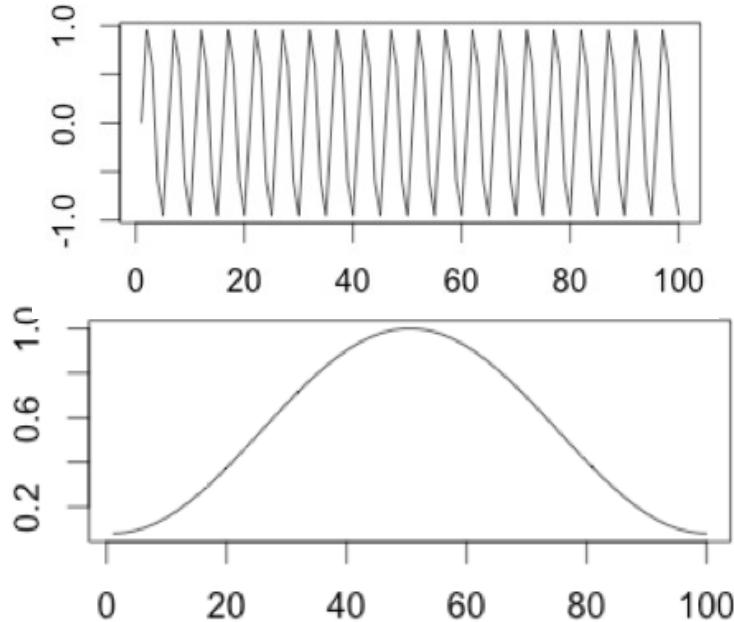
The windowed signal



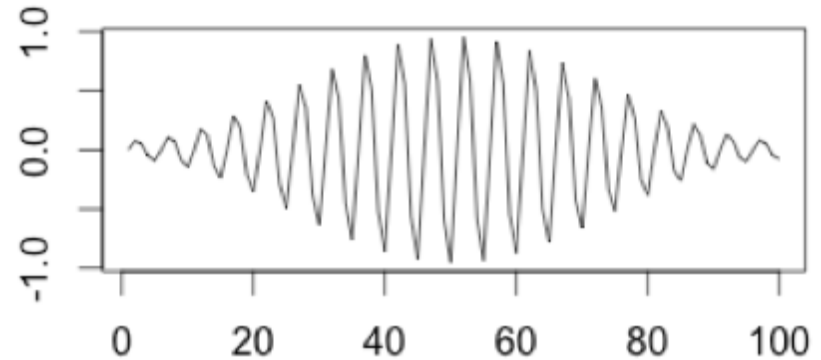
```
t = np.arange(0, 1, 1/1000)
s = sin(2*pi*10.5*t)
w = np.hamming(1000)
plot(w*s)
```

Windowing process

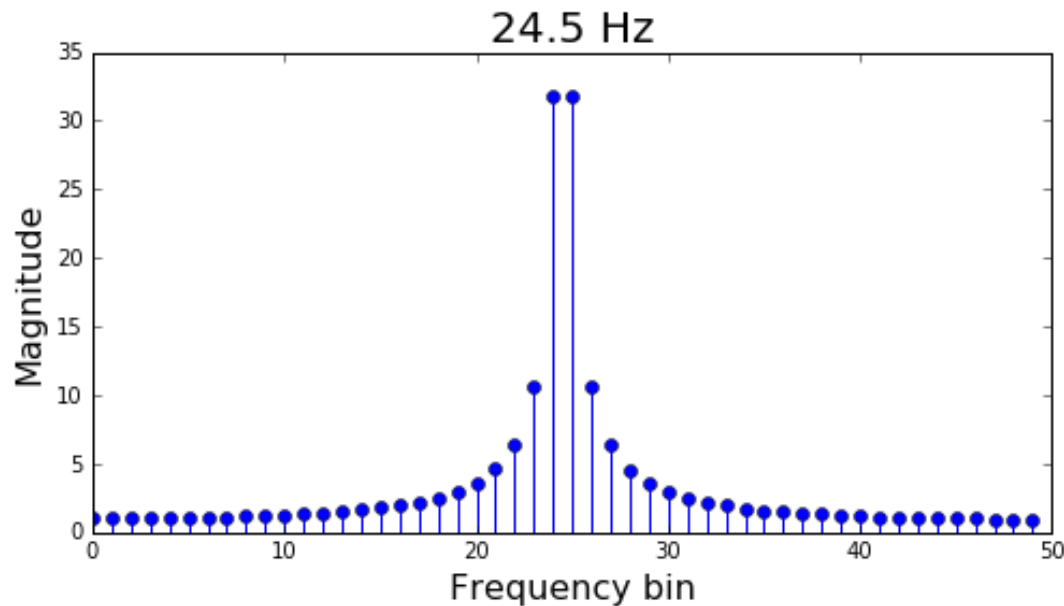
1. point-wise multiply signal with window:



2. Then apply FFT to the result



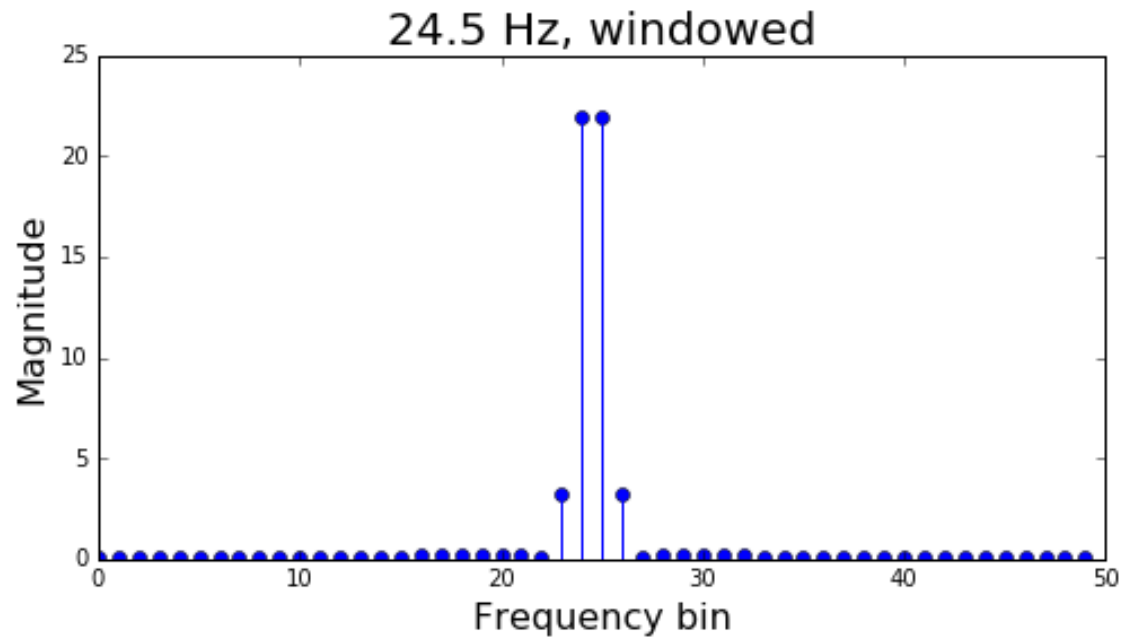
Without windowing:



```
t = np.arange(0, 1, 1/100)
s1 = sin(2*pi*25*t)
s2 = sin(2*pi*26*t)
s3 = sin(2*pi*25.5*t)
stem(abs(fft.fft(s3))[0:50])
```

leakage!!

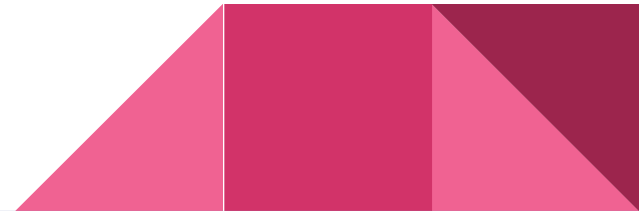
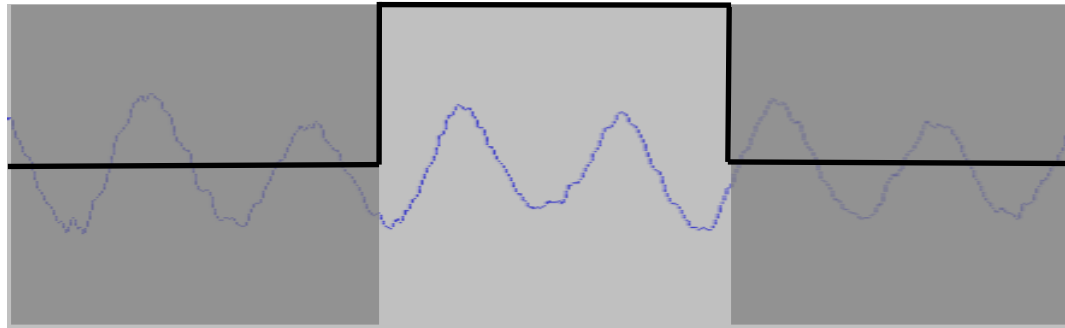
With windowing:



```
s3 = sin(2*pi*24.5*t)
windowed = np.hamming(100) * s3
stem(abs(fft.fft(windowed))[0:50])
```

You've been windowing without knowing it...

“Selecting” N time-domain samples is like point-by-point multiplication with a rectangular function (“window”):



Recall:

Convolution in the time domain is equivalent to multiplication in the frequency domain.

Also: Multiplication in the time domain is equivalent to convolution in the frequency domain.

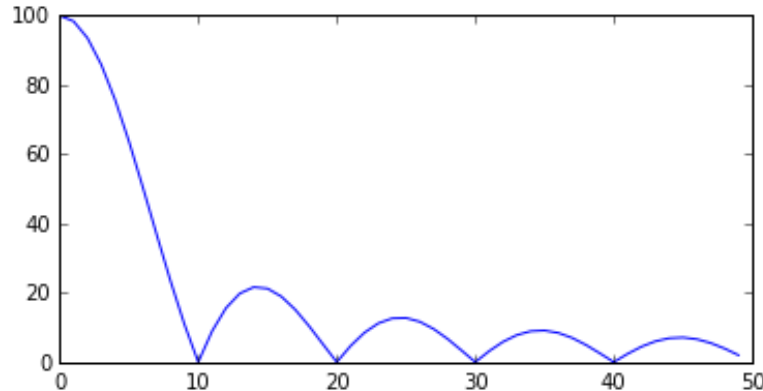
Windowing

A rectangular signal has a very “messy” spectrum!

Signal:

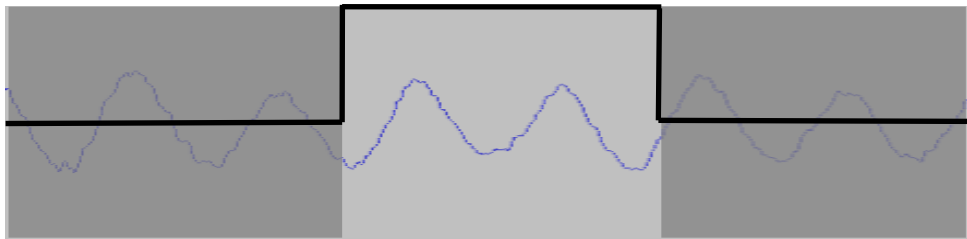


Spectrum:

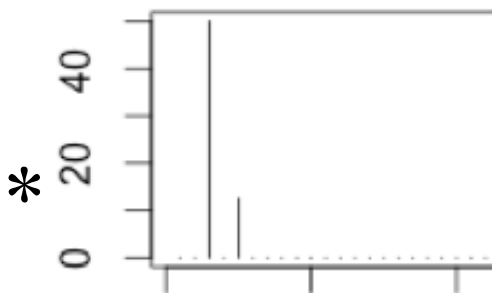
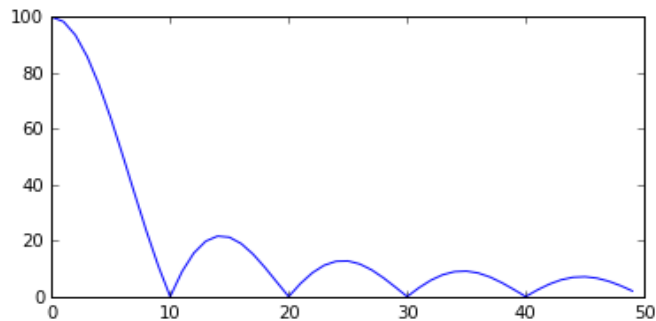


Windowing

Multiplying a signal by a rectangle in time...



is equivalent to convolving their spectra:

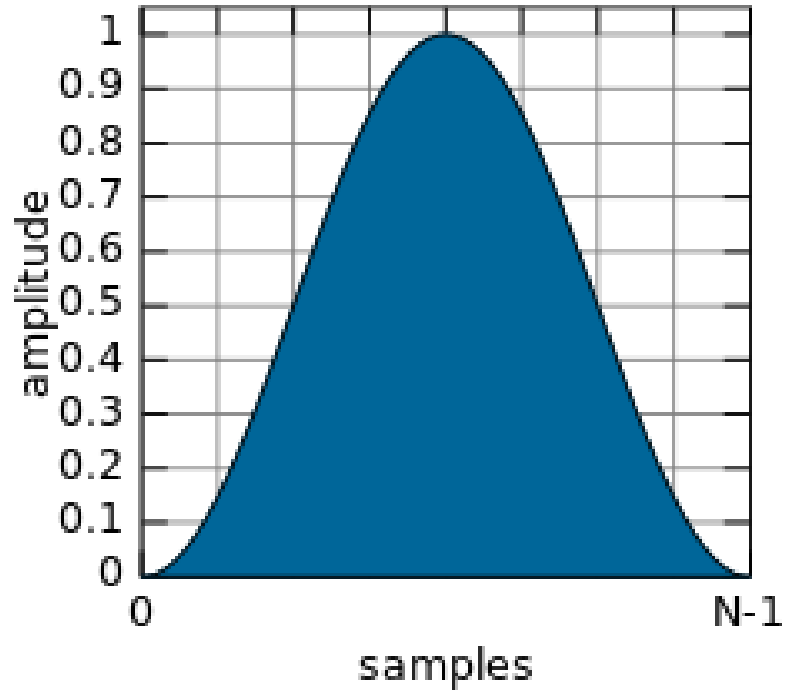


*

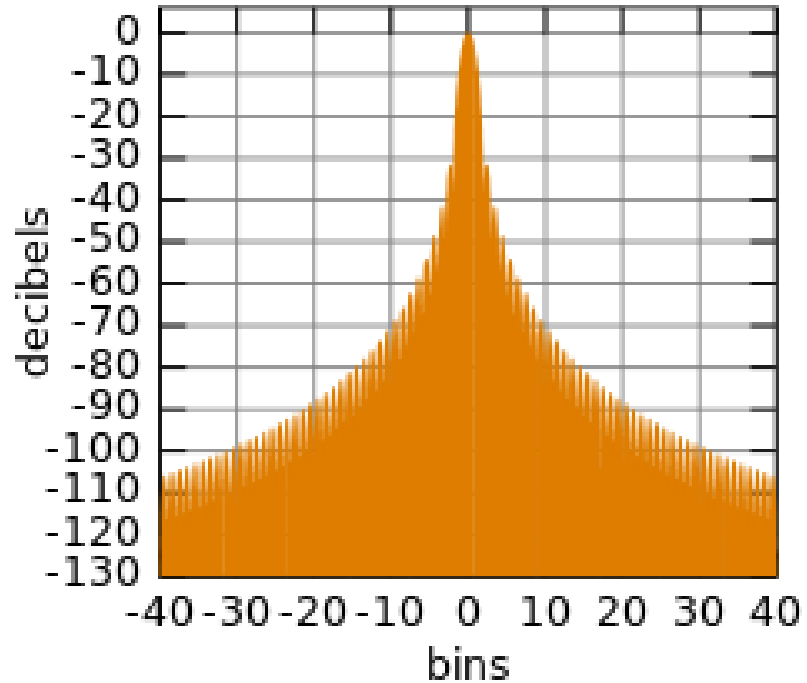


Example windows

Hann window



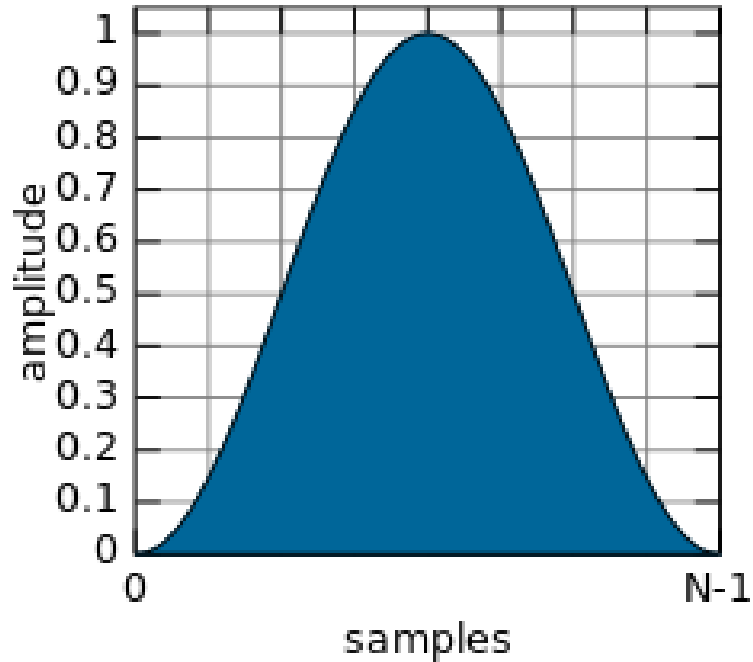
Fourier transform



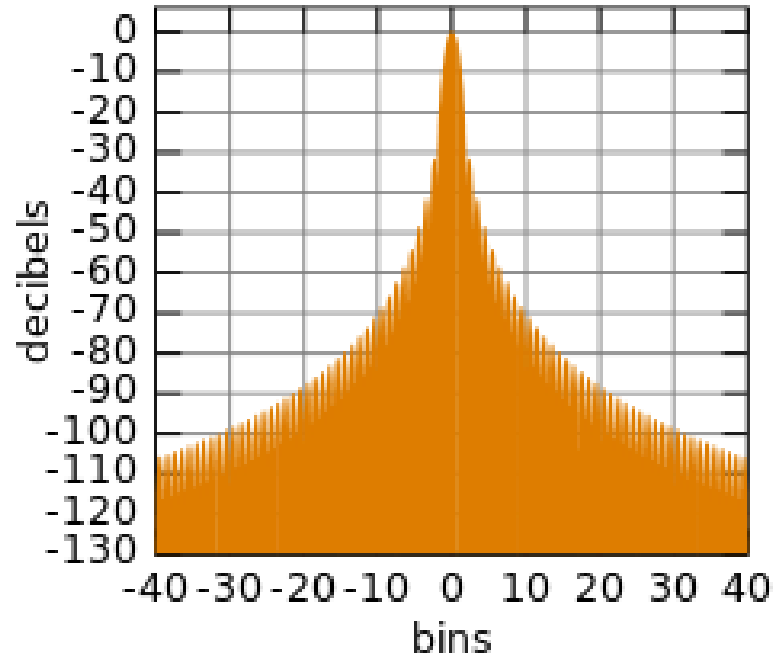
From http://en.wikipedia.org/wiki/Window_function

Example windows

Hann window



Fourier transform

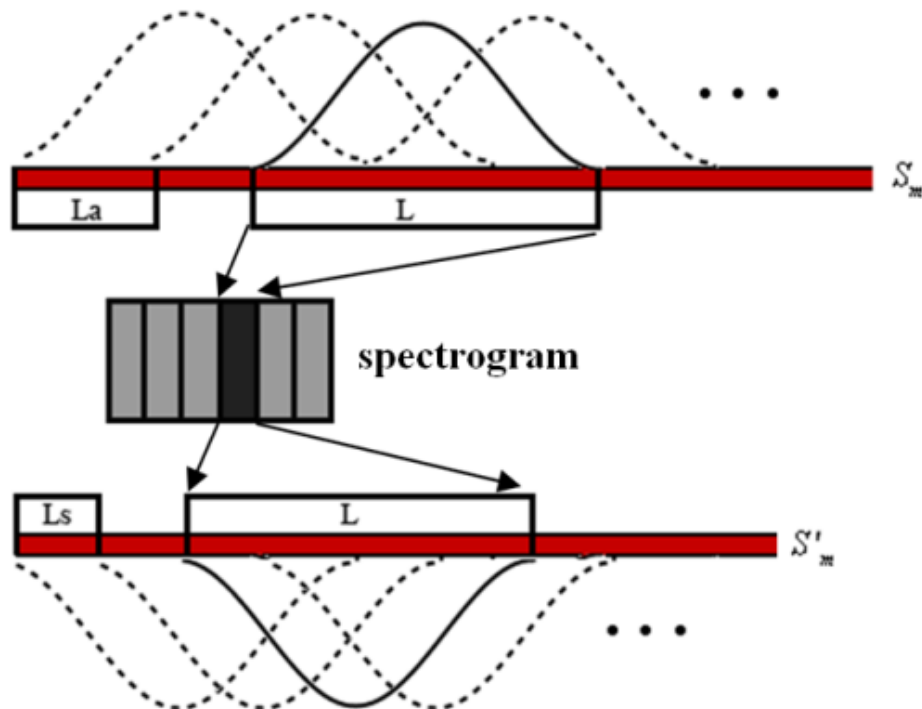


From http://en.wikipedia.org/wiki/Window_function



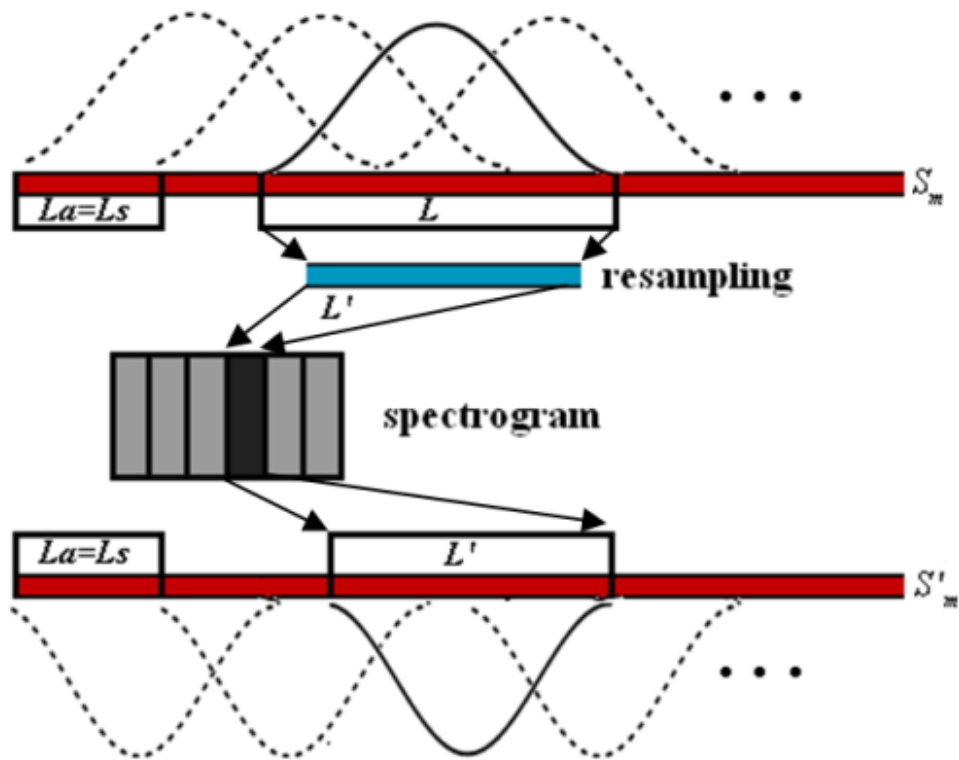
Using these ideas to
design new effects...

Time stretching: one approach



Use different
hop size in input
vs output

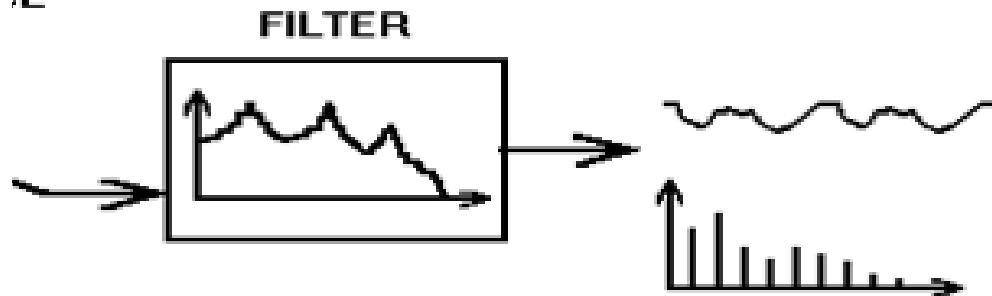
Pitch shifting: one approach



model & cross-synthesis



;E



Demo:
guitar source spectrum shaped by voice spectrum

```

#cross-synthesis code
#first, load (or synthesize) sounds into file1 and file2

newSound = zeros(size(file1))
i = 0
j = 0;
width = 2048;
hop = 2048/16;
win = np.hamming(2048)

while (i+ width < size(file1)) :
    if (j + width >= size(file2)) :
        j = 0
    frame1 = win*file1[i:i+width]
    frame2 = win*file2[j:j+width]
    f1 = fft.fft(frame1)
    f2 = fft.fft(frame2)
    ms = abs(f2)
    frame3 = f1 * ms

    sig1 = fft.ifft(frame3).real
    newSound[i:i+width] = newSound[i:i+width] + sig1
    i = i + hop
    j = j + hop

newSound = 0.7*newSound/(max(abs(newSound)))
play(newSound)
plot(newSound)

```

To read more

Pitch shifting & time stretching:

<http://blogs.zynaptiq.com/bernsee/time-pitch-overview/>

Description of signal processing in the phase vocoder:

<https://www.eumus.edu.uy/eme/ensenanza/electivas/dsp/representaciones/PhaseVocoderTutorial.pdf>

