CS 583– Computational Audio -- Fall, 2021

Wayne Snyder Computer Science Department Boston University

Lecture 09

Conclusions on Auto-Correlation for Pitch Detection Discrete Sine Transform Complex Numbers

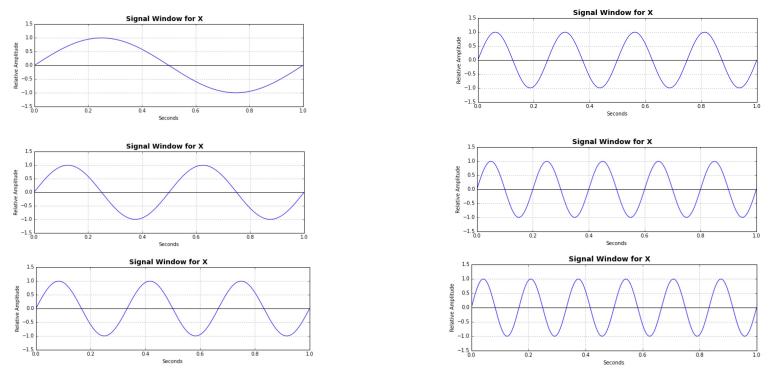
If times: The Discrete Fourier Transform



Computer Science



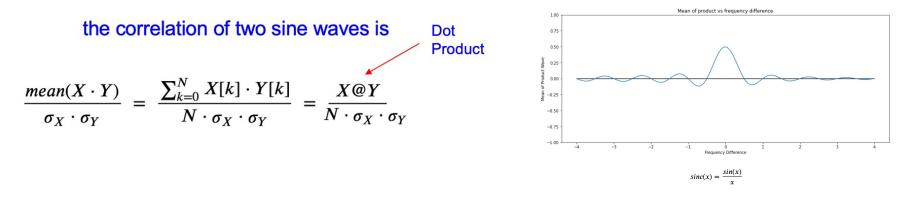
Define: For a signal X of length W samples (a "window") a window frequency is one whose period P is such that W = P * k for some integer k, i.e., an integral number of periods exactly fit within the window; alternately, it begins and ends at same instantaneous phase.



We will use these signals as **probe waves** to analyze a musical signal and assume that all such probe waves (for now) start at phase 0.0.



Recall that when we take the correlation of two sine waves, we get 1.0 if the waves are the same frequency and phase, and close to 0.0 otherwise.



However, if we suppose both waves have amplitude 1.0, we can simplify:

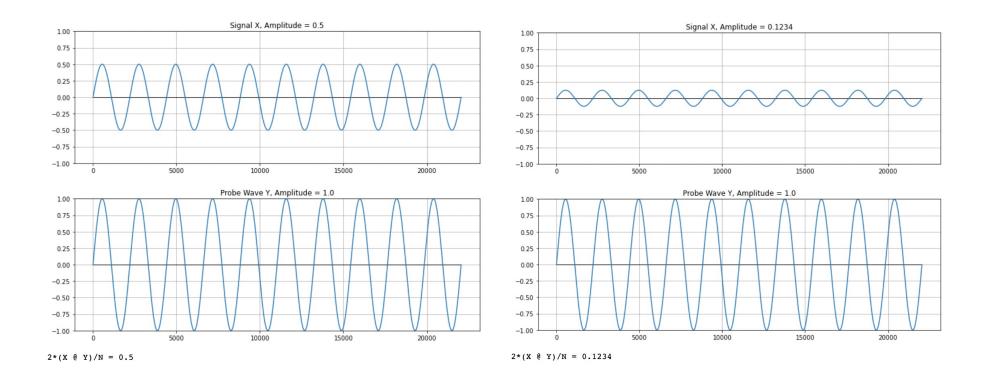
$$2 \cdot \frac{X@Y}{N} = 1.0$$

and if one (the "probe wave") has amplitude 1.0 and the other has amplitude A, where both have the same frequency and phase, we have:

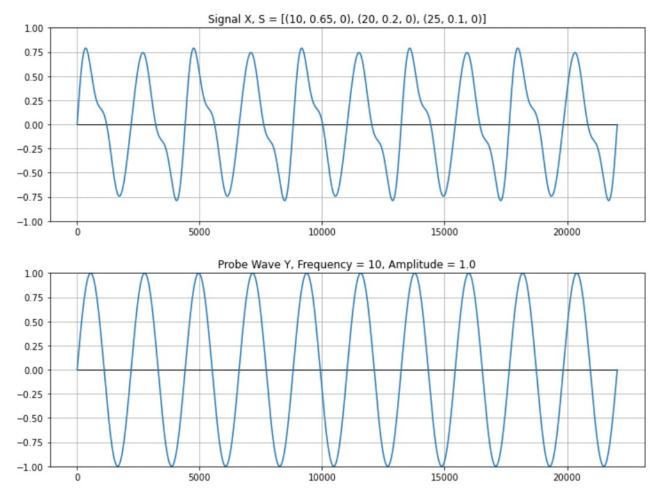
$$2 \cdot \frac{A \cdot X@Y}{N} = A\left(2 \cdot \frac{X@Y}{N}\right) = A$$



Therefore we have a "detector" for finding the amplitude of a given signal X, as long as we know the frequency and phase:

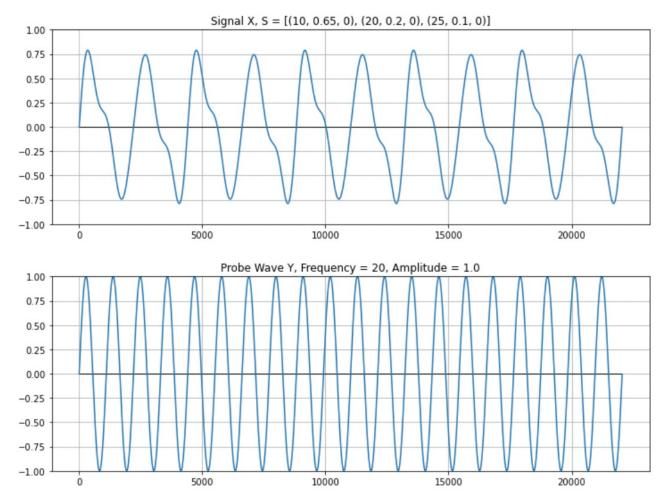


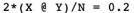




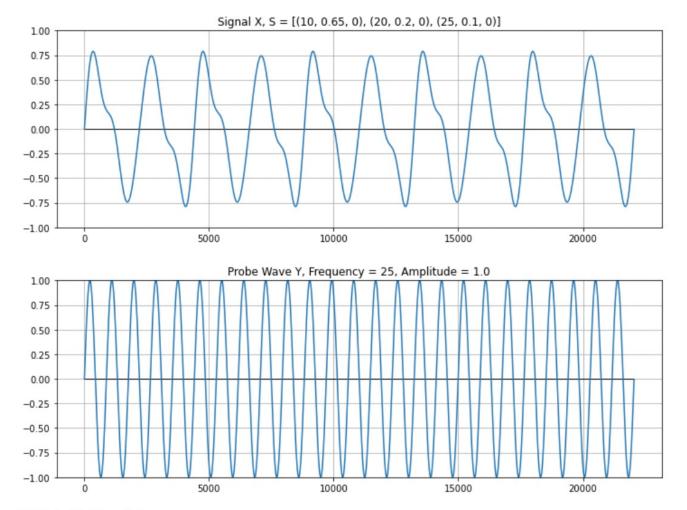
2*(X @ Y)/N = 0.65





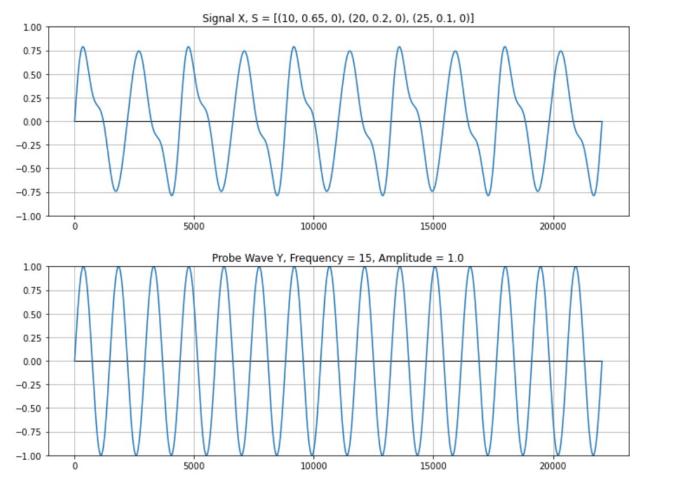






2*(X @ Y)/N = 0.1





Punchline:

Window frequencies are orthogonal, and so the probe wave can detect components of waves just as easily as simple sine waves!

2*(X @ Y)/N = -0.0



Doing this consistently for all window frequencies gives us the Discrete Sine Transform:

```
def DST(X):
    W = len(X)
                               # window length in samples
    S = [0] * (W // 2)
    for f in range(W//2): # for each probe wave f in [0..W//2]
        for k in range(W): # sum the products of signal and probe and save in S[f]
            S[f] += X[k] * sin(2 * pi * f * k / W)
        S[f] = 2 * S[f] / W # normalize to mean of products to get actual amplitude
    return S
X = makeSignal([(3.0,0.5,0.0), (5.0,0.3,0.0), (10,0.2,0.0)], 1.0)
S = DST(X)
         S[0]: 0.0
         S[1]: 7.89649143503e-12
         S[2]: -7.39108746491e-12
         S[3]: 0.49999999998
         S[4]: 4.00080251115e-12
         S[5]: 0.29999999999
         S[10]: 0.2000000000
         S[11]: -1.83215982068e-12
                                                                                  9
         S[22049]: 2.55635210657e-12
```



Doing this consistently for all window frequencies gives us the Discrete Sine Transform:

Note:

The transform can ONLY detect window frequencies = k * f for f = 1 / W (in secs) = k * SR / W (in samples)

So a window of 1.0 seconds can detect 0, 1, 2, ..., 22049 ONLY of 0.1 seconds can detect 0, 10, 20, 30, ..., 22040 of 0.2 seconds can detect 0, 5, 10, ..., 22040

Another problem is that this took 20minutes to run!

Double for loop with W = 44100... 44100 * 22050 = 972,405,000 executions of inner loop!



Doing this consistently for all window frequencies gives us the **Discrete Sine Transform**:

X = makeSignal([(30.0,0.5,0.0), (50.0,0.3,0.0), (100.0, 0.2,0.0)], 0.1) S = DST(X)

Bin	Amp	Freq		
S[0] :	0.0	0		
S[1]:	-3.93616764277e-12	10		
S[3] :	0.499999999987	30		
S[4]:	6.72379914407e-12	40		
S[5] :	0.3000000001	50		
S[10]: 0.29999999997 100				
S[2204]: 4.73093370979e-13 22040				

This took about 15 seconds to run



Doing this consistently for all window frequencies gives us the **Discrete Sine Transform**:

X = makeSignal([(30.0, 0.5, 0.0), (50.0, 0.3, 0.0), (100.0, 0.2, 0.0)], 0.2) S = DST(X)

This took about 1 minute to run

Bin S[0]:	Amp 0.0	Freq
	9.58745925935e-13	5
S[6]:	0.5	30
S[10] :	0.3000000001	50
S[20]:	0.29999999999	100
S[440	9]: 7.23056298634e-12	22045

Digital Audio Fundamentals: The Discrete Sine Transform



Returns a spectrum of amplitudes (in range -1 .. 1)

$$S = [A_0, A_1, A_2, \dots, A_{N/2-1}]$$
 assuming w is even

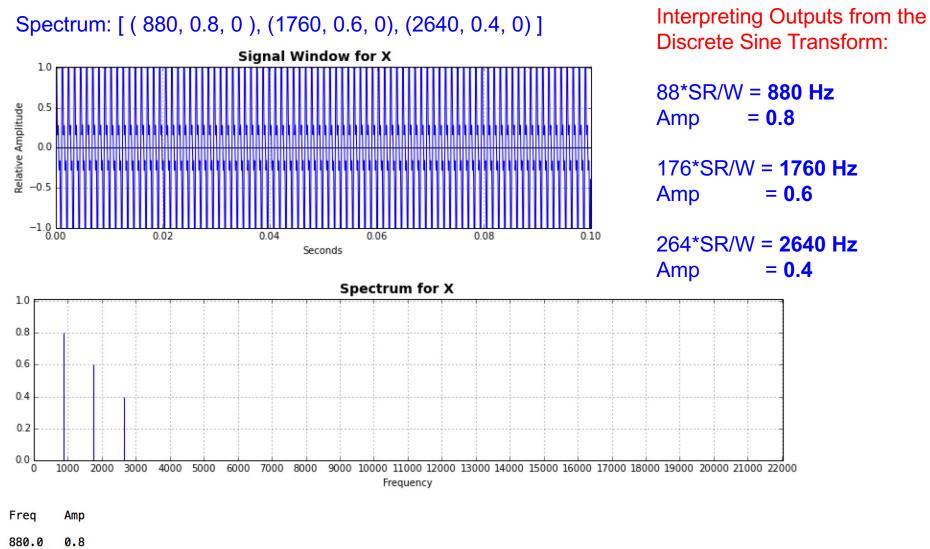
for window frequencies

 $W_f = [0, 1, 2, ..., N//1 - 1]$

and actual frequencies

 $F = [0, 1R, 2R, ..., R^*(N/2 - 1)]$ for R = SampleRate / W

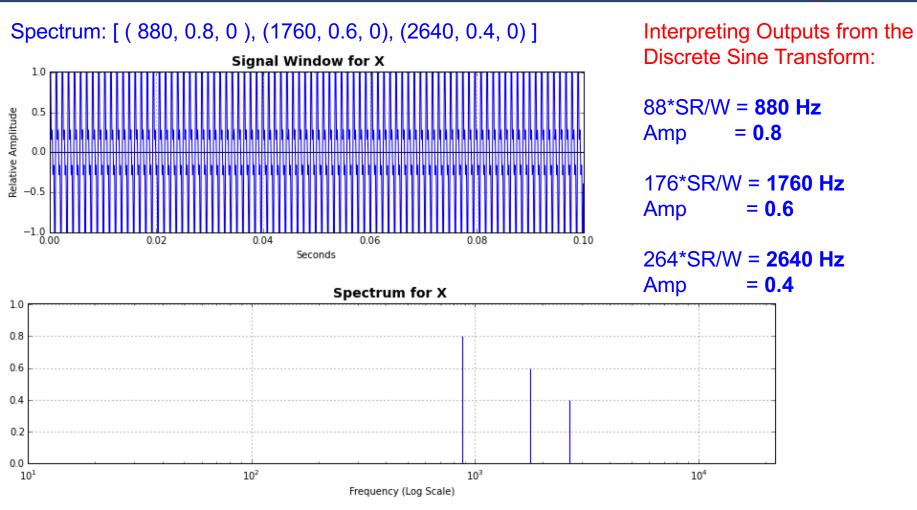




1760.0 0.6

2640.0 0.4



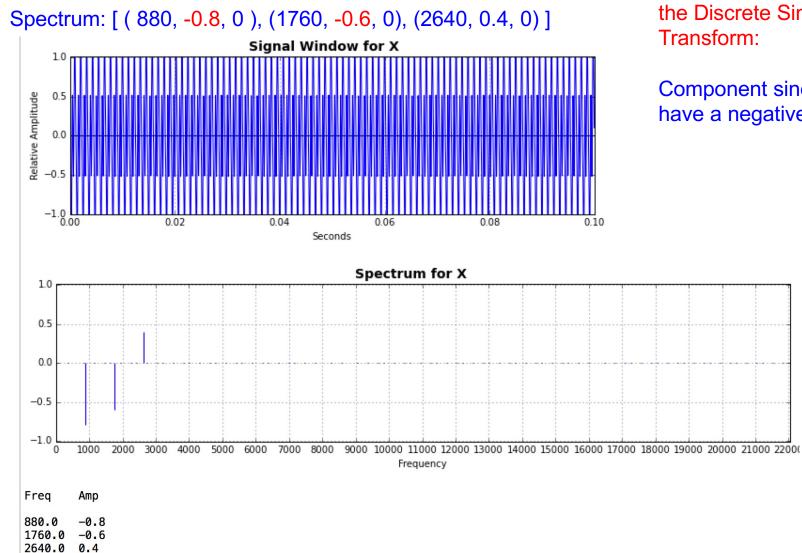


Freq Amp

880.0 0.8 1760.0 0.6

2640.0 0.4



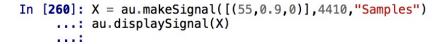


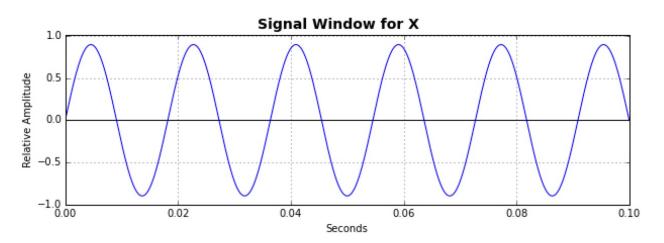
Interpreting Outputs from the Discrete Sine

Component sine waves may have a negative amplitude.

Digital Audio Fundamentals: The Discrete Sine Transform

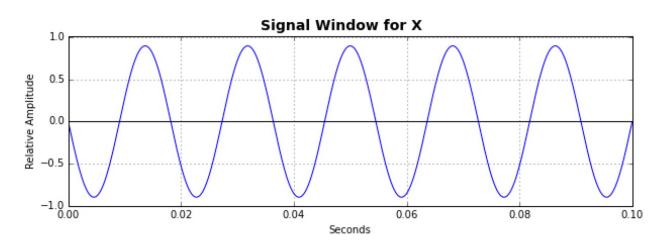






Component sine waves may have a negative amplitude; they will produce the negative of a squared wave, and report negative amplitudes just as they report positive amplitudes.

```
In [261]: X = au.makeSignal([(55,-0.9,0)],4410,"Samples")
    ...: au.displaySignal(X)
    ...:
```



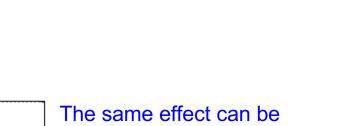
Digital Audio Fundamentals: The Discrete Sine Transform

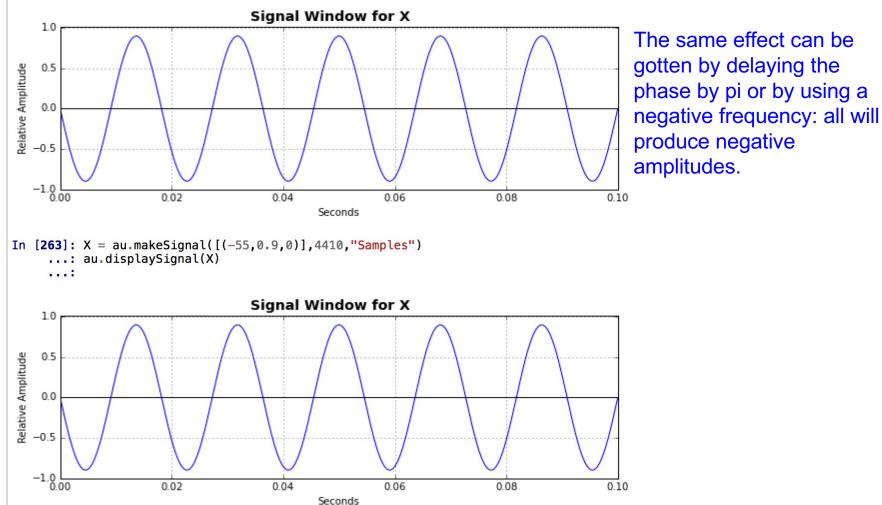
In [262]: X = au.makeSignal([(55,0.9,pi)],4410,"Samples")

...: au.displaySignal(X)

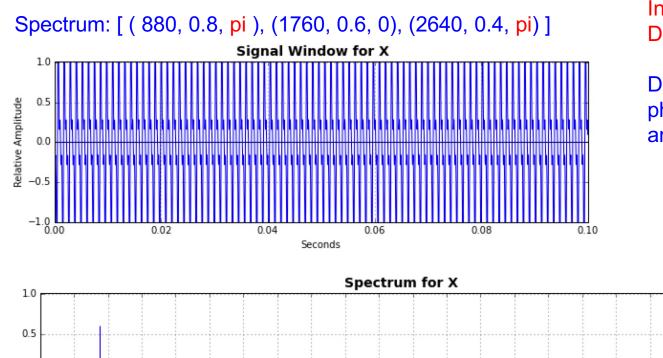
.....





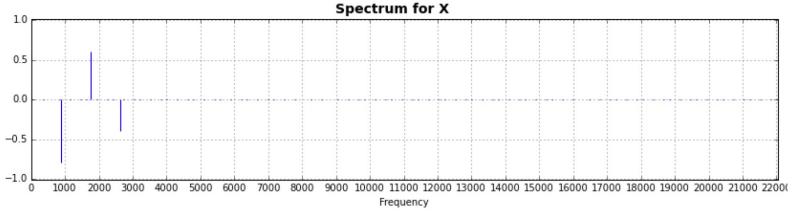






Interpreting Outputs from the Discrete Sine Transform:

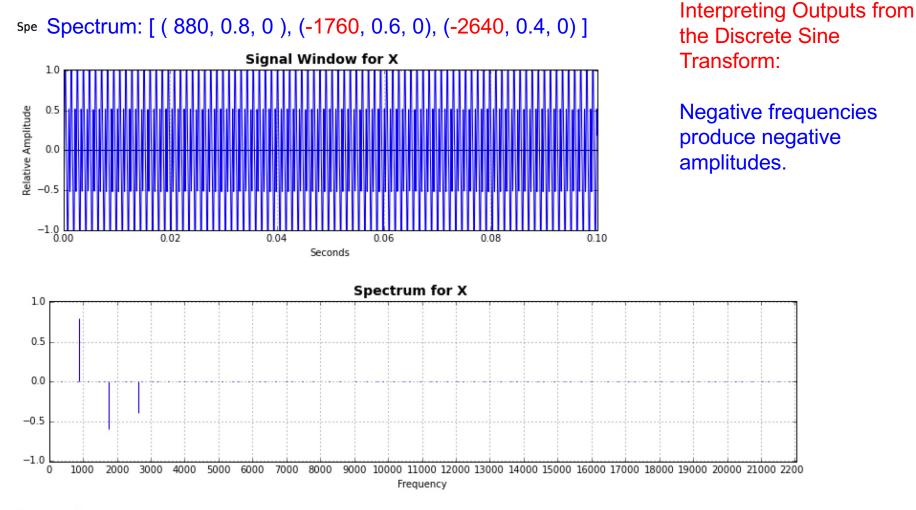
Delaying a component by phase pi produces negative amplitudes.



Freq Amp

880.0 -0.8 1760.0 0.6 2640.0 -0.4

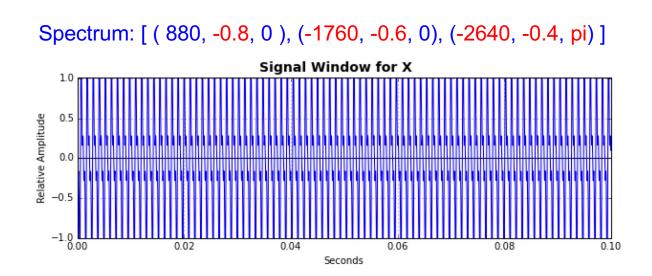




Freq Amp

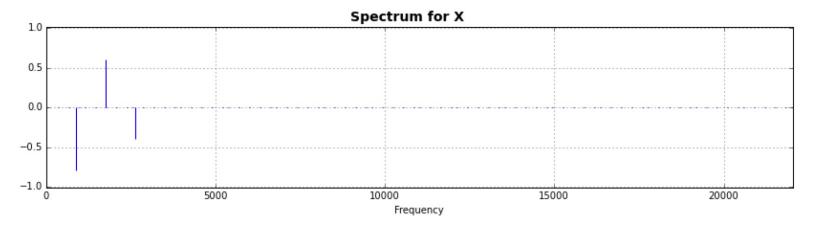
880.0 0.8 1760.0 -0.6 2640.0 -0.4





Interpreting Outputs from the Discrete Sine Transform:

Doing combinations of these will flip the amplitude back and forth:



Freq Amp

880.0 -0.8 1760.0 0.6 2640.0 -0.4



There are three problems (so far):

- (1) This is horribly inefficient: O(N^2) for N = len(X)
 - ✓ Solution: There is a fast version of the transform, the Fast Fourier Transform (FFT), based on a recursive algorithm, which runs in O(N log(N)).

(2) The resolution is limited to multiples of f = SR / W (in samples)

X No solution, unfortunately, can try different window sizes, but stuck with this!

(3) All components and probe waves have to be at the same phase (e.g., 0.0)

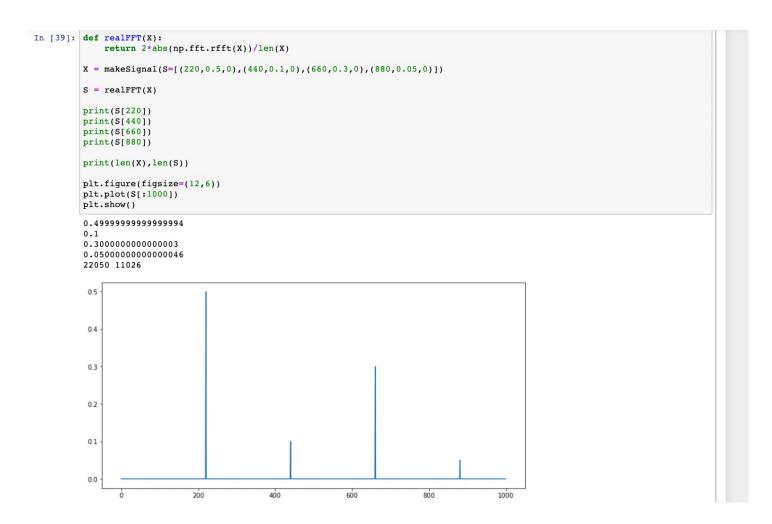
✓ Solution: If we do all the work with complex numbers, we can avoid issues of phase

A brief summary of Complex Numbers on the board.....

Digital Audio: The Discrete Fourier Transform



I have provided in the Intro Notebook an implementation of the FT which returns real results:





So, we have an efficient algorithm which does not care about phase, but problem 2 is still with us!

(2) The resolution is limited to multiples of f = SR / W (in samples)

X No solution, unfortunately, can try different window sizes, but stuck with this!

<pre>In [58] def realPT?(X): realPT?(X) realPT?(X) print('window frequency', round4(1/0.19)) print('window frequency', round4(1/0.19)) print('\$1' attribut'; *f', round4(1/0.19)) print('\$1' attribut'; *f', round4(1/0.19)) pit.figure(figsine=(12,6)) pit.pixt('\$1' attribut'; *f(i)) pit.pixt('aff', *tribut'; *f(i)) pit.pixt('aff', *tribut'; *f(i)) pit.pixt('aff', *tribut'; *f(i)) pit.pixt('aff', *tribut'; *f', *f(i)) pit.pixt('aff', *f', *f(i)) pit.pixt('aff', *f', *f(i)) pit.pixt('aff', *f', *f(i)) pit.pixt('aff', *f', *f(i)) pit.pixt('aff', *f', *f', *f(i)) pit.pixt('aff', *f', *f', *f(i)) pit.pixt('aff', *f', *f', *f', *f', *f', *f', *f',</pre>		
<pre>S = realFFT(X) print(*window frequency*, round4(1/0.19)) print(222 =*,220/(1/0.19), ***, round4(1/0.19)) print() for i In range(35,50); print('S[**etr(i)*']:*,5[i]) plt.figure(figsize-(12,6)) plt.plot(range(s),5;0),(33:50)) plt.abov() vindow frequency 5,2632 220 = 41.8 * 5,2632 S(35): 0.013844199139476594 S(35): 0.013845107350413 S(35): 0.0139490193120572 S(40): 0.052990712802735976 S(41): 0.1996/S0173150195 S(41): 0.03592160013813 S(45): 0.0359215066 S(47): 0.013531516616080727 04 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</pre>	In [58]:	<pre>def realFFT(X): return 2*abs(np.fft.rfft(X))/len(X)</pre>
<pre>print(*window frequency*, round4(1/0.19)) print(*220 =*,220/(1/0.19), ***, round4(1/0.19)) print(*g(*qt=r(1),*); *,S(1)) plt.figure(figise=(12,6)) plt.plot(range(35,50),S(35550)) plt.show() vidow frequency 5.2622 220 = 41.8 * 5.2632 S(35): 0.01384199359476984 S(36): 0.015263134271509 S(37): 0.0156931066106816 S(36): 0.052290072882735976 S(40): 0.05290072882735976 S(40): 0.0529007288773966 S(41): 0.01399078463109093 S(41): 0.023967243171266 S(43): 0.0735350075791623 S(44): 0.023967243171266 S(44): 0.015531516616080727 04 04 04 04 04 04 04 04 04 04 04 04 04</pre>		<pre>X = makeSignal(S=[(220,0.5,0)],duration = 0.19)</pre>
<pre>print(*220 *, 220/(1/0.19), ***, round4(1/0.19)) print() for i in range(35,50): print(*2(**tr(1)*):*,S(1)) plt.figure(figise(12,6)) plt.show() vindow frequency 5.2632 20 = 41.8 * 5.2632 3(35): 0.013844199359476984 5(36): 0.0265201334271509 5(36): 0.0269201378610816 5(39): 0.0269201378610816 5(39): 0.039097288233976 5(41): 0.01392907288233976 5(41): 0.0139297588173664 5(43): 0.03957588173664 5(44): 0.045522153519664 5(45): 0.0300743601890943 5(46): 0.026927284771206 5(47): 0.0155151616000277 04 04 03 04 04 04 04 04 04 04 04 04 04 04 04 04</pre>		S = realFFT(X)
<pre>print() for i in range(35,50): print('S[**str(1)**]:*,S[1]) plt.figure(figure(2,6)) plt.abov() vindow frequency 5.2632 S(35]: 0.013844199359476984 S(36]: 0.01626521334271599 S(37): 0.0139697187235076 S(30): 0.0239401016510803 S(30): 0.0239407180275975 S(30): 0.0239407180275975 S(40): 0.0239907280275975 S(40): 0.0239907280275976 S(41): 0.013952160108013 S(45): 0.0030738601890943 S(45): 0.01395216013803 S(45): 0.013953156616000727 04 03 04 03 04 04 03 04 04 04 04 04 04 04 04 04 04 04 04 04</pre>		<pre>print("window frequency", round4(1/0.19))</pre>
<pre>print("\$["*#tr(i)*"]:",8[1]) plt.figure(figsizee(12,6)) plt.plot(range(35,50),8[35:50)) plt.show() window frequency 5.2632 20 = 41.8 * 5.2632 \$[35]: 0.013844199359476984 \$[36]: 0.01626521334271509 \$[37]: 0.0196978273507413 \$[38]: 0.024945011066108e16 \$[3]]: 0.03299087288273597 8[40]: 0.05299087288273597 8[40]: 0.05299087288273597 8[40]: 0.0755360975791623 8[41]: 0.1755360975791623 8[44]: 0.0452982591498138 8[44]: 0.045292867234771206 8[41]: 0.01862382559498388 \$[44]: 0.013531516616080727 </pre>		
<pre>plt.pldt(riange(15, 50), s[35:50]) plt.show() window frequency 5, 2632 220 = 41.8 * 5, 2632 S(35]: 0.01384199359476984 S[35]: 0.01384199359476984 S[37]: 0.013659578273507413 S[37]: 0.0296987282735076 S[41]: 0.045997528173576 S[41]: 0.01394908108103 S[44]: 0.07955360975791623 S[44]: 0.04359291600138813 S[45]: 0.03007436801890943 S[44]: 0.04359291600138813 S[44]: 0.013531516616080727 044 03 04 03 04 03 04 03 04 03 04 03 04 03 04 03 04 03 04 03 04 03 04 03 04 04 03 04 04 03 04 04 03 04 04 03 04 04 03 04 04 03 04 04 03 04 04 03 04 04 04 04 04 04 04 04 04 04 04 04 04</pre>		
220 = 41.8 * 5.2632 S(35): 0.013844199359476984 S(37): 0.01969878273507413 S(38): 0.02494501106610816 S(39): 0.03394908193120572 S(41): 0.01352909572882735976 S(41): 0.01352909572892735976 S(41): 0.01352921500138813 S(45): 0.030743601890943 S(45): 0.030743601890943 S(45): 0.030743601890943 S(45): 0.013529246771206 S(47): 0.013529246973838 S(48): 0.01566654011703157 S(49): 0.013531516616080727		plt.plot(range(35,50), s[35:50])
S [36] : 0.01626521334271509 S [37] : 0.01626521334271509 S [39] : 0.024945011066108816 S [39] : 0.0359908173120572 S [40] : 0.052990872882735976 S [41] : 0.199673431300199 S [42] : 0.46639975281719664 S [44] : 0.04359291600138013 S [45] : 0.0209672244771206 S [47] : 0.015666854011703157 S [49] : 0.015666854011703157 S [49] : 0.015666854011703157 S [49] : 0.015661616080727		window frequency 5.2632 220 = 41.8 * 5.2632
S [40]: 0.052990872882735976 S [41]: 0.1994678431300199 S [42]: 0.46639975281719664 S [43]: 0.07955309779791623 S [44]: 0.04359291600138813 S [45]: 0.0209672244771206 S [47]: 0.015666854011703157 S [49]: 0.015666854011703157 S [49]: 0.015661616080727		\$[36]: 0.01626521334271509 \$[37]: 0.01969878273507413 \$[38]: 0.024945011066108816
S [42] : 0.46639975281719664 S [43] : 0.0795509779701623 S [44] : 0.04359291600138013 S [45] : 0.02299672244771206 S [47] : 0.0186238255949838 S [48] : 0.015666854011703157 S [49] : 0.013531516616080727 04- 03- 04- 04- 04- 04- 04- 04- 04- 04		S[40]: 0.052990872882735976
S[44]: 0.04359291600138013 S[45]: 0.02299672244771206 S[47]: 0.0186232825949838 S[48]: 0.015666854011703157 S[49]: 0.013531516616080727		S[42]: 0.46639975281719664
S[46]: 0.02299672244771206 S[47]: 0.0186232825949338 S[48]: 0.015666854011703157 S[49]: 0.013531516616080727		s[44]: 0.04359291600138813
S[48]: 0.015666854011703157 S[49]: 0.013531516616080727		S[46]: 0.02298672244771206
		S[48]: 0.015666854011703157
		04
		02-
		01-
36 38 40 42 44 46 48		
		36 38 40 42 44 46 48

More on this next time!