Lecture 2

FOURIER TRANSFORMS

Change of Basis

Let's start with 'why'. The cheap answer is 'to get a different perspective; to see things that couldn't be seen otherwise'.

Or, we could just do an example. Eigenvalues and eigenvectors were made for this. Let's look at the matrix $T = \begin{pmatrix} 5 & 1 \\ 1 & 5 \end{pmatrix}$. Since we'll be changing bases, we ought to specify that the matrix is taken with respect to the standard basis of $\mathbf{R}^2$, $\mathbf{s_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$; $\mathbf{s_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.Then, as always, the first column of T is $T\mathbf{s_1}$, and the second is $T\mathbf{s_2}$.

Problem Show that T has eigenvalues $\lambda_1 = 6$; $\lambda_2 = 4$, and that the corresponding eigenvectors may be taken to be

$$\mathbf{b_1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \ \mathbf{b_2} = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

Note that the two are orthonormal. Of course the change-of-basis map from the $\mathbf{s}$ to the $\mathbf{b}$ basis is

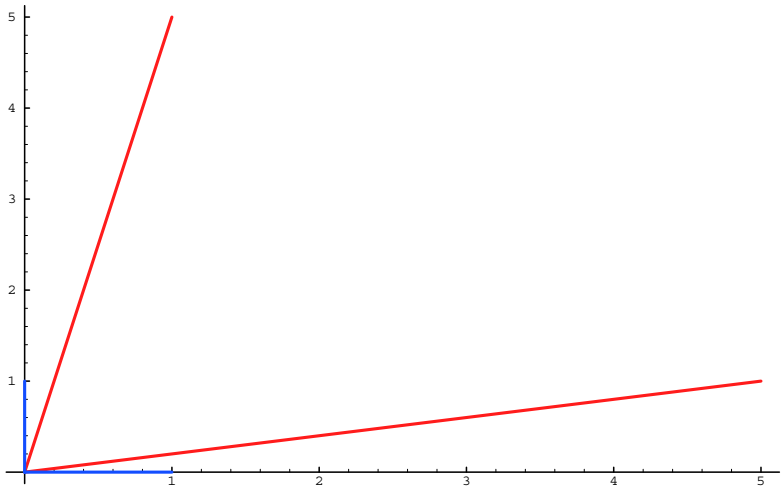
$$C_{\mathbf{sb}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$$

Prove that the inverse of $C_{\mathbf{sb}}$, that is, $C_{\mathbf{bs}}$, is given by

$$C_{\mathbf{bs}} = C_{\mathbf{sb}}^{-1} = C_{\mathbf{sb}}^{*}$$

where we use the * to denote complex-conjugate and transpose. Now, show that the matrix T, in the $\mathbf{b}$ basis, is $C_{\mathbf{bs}} T C_{\mathbf{sb}} = \begin{pmatrix} 6 & 0 \\ 0 & 4 \end{pmatrix}$, which is what we'd expect.

In this 'perspective', the matrix T just stretches the basis differently in the two orthogonal directions. Compare the picture below, showing the action of T on the standard basis.



Homework Problem As we'll be changing bases rather a lot in this course, let's prove some basic results. Let V be a finite-dimensional inner product space. Given any pair of orthonormal bases **b, s** for V, show that $C_{\mathbf{bs}} = C_{\mathbf{sb}}^{-1} = C_{\mathbf{sb}}^{*}$. Moreover, show that the change of bases maps are *unitary*, that is, that for any pair of vectors $x\ y \in V$,

$$(C_{\mathbf{bs}}x, C_{\mathbf{bs}}y) = (x,\ y)$$

That is, unitary transformations preserve inner products. As angles and lengths are defined in terms of inner products . . .

If $V = \mathbf{R}^n$ with the usual inner product, show that if $x = (x_1,\ x_2,\ \ldots\ ,\ x_n)$ in the **s** basis, but $x = (w_1,\ w_2,\ \ldots\ ,\ w_n)$ in the **b** basis, then

$$\sum |x_j|^2 = \sum |w_j|^2$$

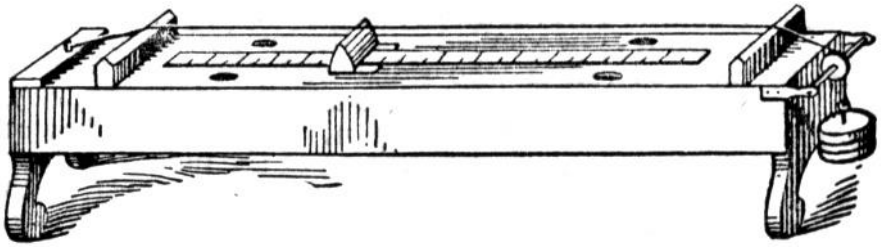
Lecture 2

FOURIER TRANSFORMS

Bases of Exponentials

The bases we chose for Fourier transforms on $l^2(\mathbf{Z}), L^2(\mathbf{Z_K}), L^2(\mathbf{R}), L^2[0,\ 1]$ are all bases of exponentials. We want to discuss why complex exponentials might be reasonable choices, and we'll do this by looking at special characteristics the exponentials have for the three distinct function spaces.

The first space we'll look at is $L^2[0,\ 1]$. The exponential $e^{-2\pi inx}$ can be written as $e^{-2\pi inx} = \cos(2\pi nx) + i\sin(2\pi nx)$, so that we are really dealing with sines and cosines of different frequencies on $[0,\ 1]$. You can see why both sines and cosines are needed to represent a function on $[0,\ 1]$; $\sin(2\pi nx)$ is zero at $x = 0,\ 1$, so sines alone can't represent general periodic functions. However, sines are important historically in the development of the theory of music in ancient Greek philosophy. The early experiments by Pythagorean philosophers were based on the sound given off by a *monochord*. The monochord consisted of a single string, held down at either end, and stretched taut so that it could vibrate. If we think of the string having length one, and $f(x)$ as representing the height of the string, then the condition that the string be help down at either end is equivalent to requiring that $f(0) = f(1)$, which sine functions are perfectly adapted to do.



Now, imagine the string plucked at time $t = 0$. Think of the plucking as an initial height of the string at each point x: $height = f(x)$ As we let go, the string will change height over time, leading to other heights, $u(x,t)$, over time. Of course we need to have $u(x,0) = f(x)$, and $u(0,t) = u(1,t) = 0$. If the string motion is relatively small, it's easy to analyze what that motion has to be over time:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

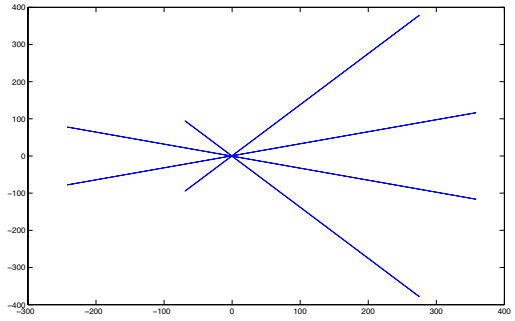
Homework Problem Show that $u(x,t) = \cos(2\pi nt)\ \sin(2\pi nx)$ solves this equation. What must c be in this case?

Notice that $u(x,0) = \sin(2\pi nx)$ is a sine wave. As time varies, u is called a *standing wave*, and it produces what is called a *pure tone* for the monochord. The case $n = 1$ is called the *fundamental* frequency (as no standing waves of lower frequencies can exist on the monochord), and the higher n are called *harmonics*. The Fourier series representation of a function then has a musical interpretation: every sound on the monochord is a sum of pure tones.

The Fourier representation of sounds produced by a monochord is nice all by itself, but it turns out that the human ear uses vibrations of (biological analogues of) strings as its basic mechanism for hearing. Thus, for humans, the Fourier basis is the correct biological basis for understanding hearing – an idea that we'll return to. For now, we remark that this fact is at the basis of the mp3 method of music compression.

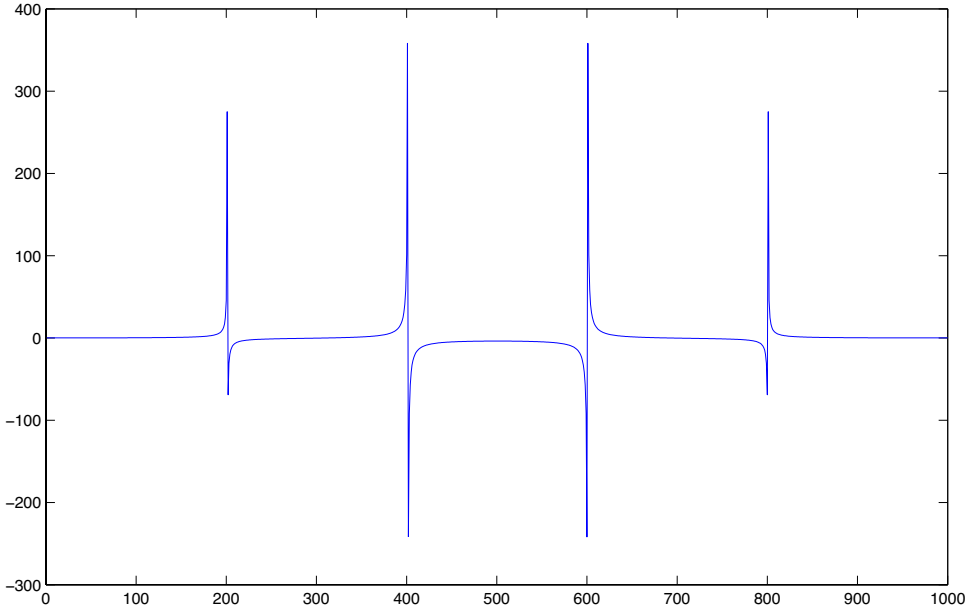
Let's see the Fourier transform in action. The computations we'll be doing are based on real signals, which means they've all been sampled at some sampling rate. This means we'll be using the Discrete Fourier Transform. As discussed in Lecture, the fastest implementation of this is the FFT. Here's what it looks like in Matlab:

```
> x=linspace(0,1,1000);
% I created a 'sample space'; this samples the interval [0, 1] a thousand times. If you think of [0, 1]
% as one second, then you have a thousand samples a second -- a kiloHertz, kHz.

> y=sin(2*pi*x*200)+sin(2*pi*x*400);
% create a sum of two frequencies: 200Hx and 400Hz.

> z=fft(y,1000);
%tells Matlab to do the FFT on 1000 points (which it will write as 2^3*5^3)

> plot(z)
```



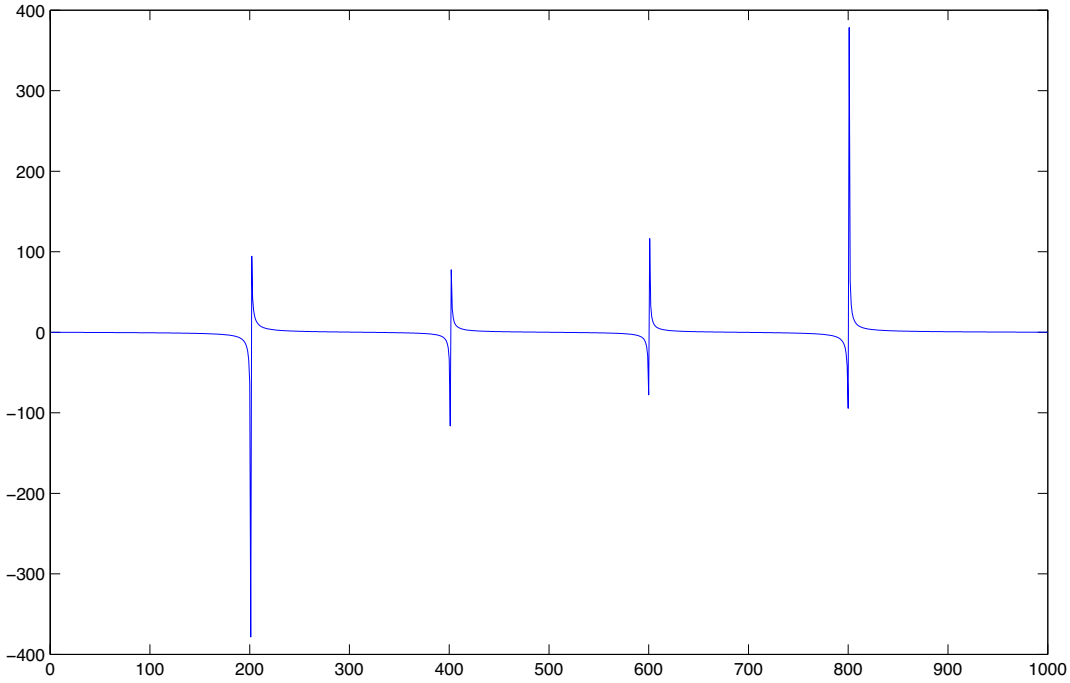
Whoa, that's weird! The problem is that z is a complex vector, with real and imaginary parts. Let's try just the real part:

```
> plot(real(z))
```



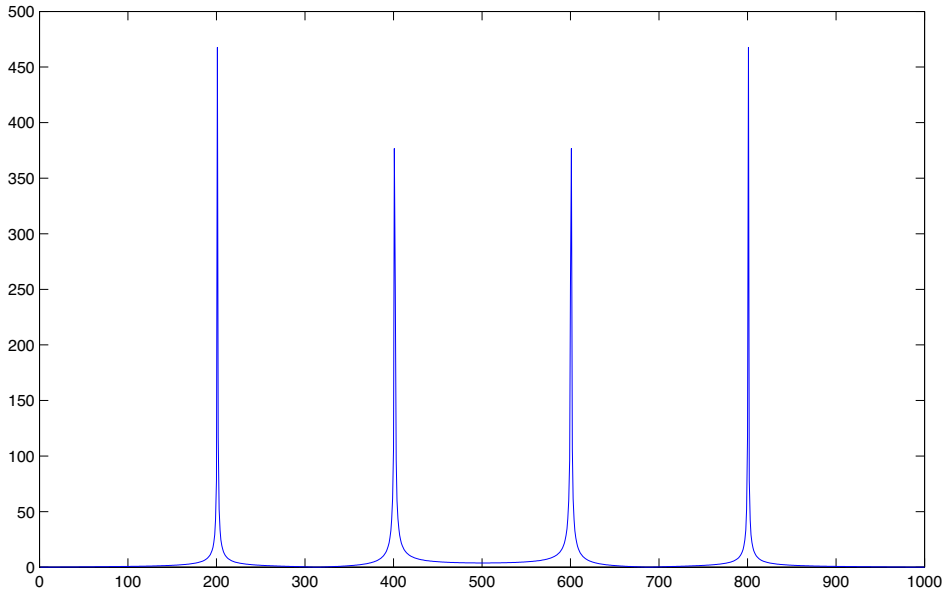
A bit better: there are peaks at 200 and at 400, but symmetric peaks at 600 and 800 as well. As there's no 600 or 800 cycle wave in my signal y, something peculiar is going on. Let's look at the imaginary part:

```
> plot(imag(z))
```



The same kind of strangeness, but this time anti-symmetric peaks at 600 and 800 as well. Put it all together by plotting the absolute value:

```
> plot(abs(z))
```



This, of course, removes the negative components of the transform completely. This kind of graph is called a *power spectrum* -- with the understanding that it's only a part of the Fourier transform, the magnitude. Whether one has negative or positive values, real or imaginary parts, that's called the *phase* of the transform.

Time to stop being mysterious. First of all, we're sampling at 1000Hz. By Nyquist, the highest frequency we can "see" is 500hz. This explains, in a way, why the frequencies from 500hz to 1000hz are duplicates of the information from 0Hz to 500Hz.

Homework Problem We can be more precise than that. If f is in $l^2(\mathbf{Z_K})$, then $\hat{f}$ is also defined on $0,\ 1, \ldots\ , K-1$. Prove that

$$\begin{aligned} \Re(\hat{f})(j) &= \Re(\hat{f})(K-j) \\ \Im(\hat{f})(j) &= -\Im(\hat{f})(K-j) \\ \left|\hat{f}(j)\right| &= \left|\hat{f}(K-j)\right| \end{aligned}$$

Because of this, it's traditional to plot only half the FFT values; typically

```
> plot(abs(z(1:500)))
```

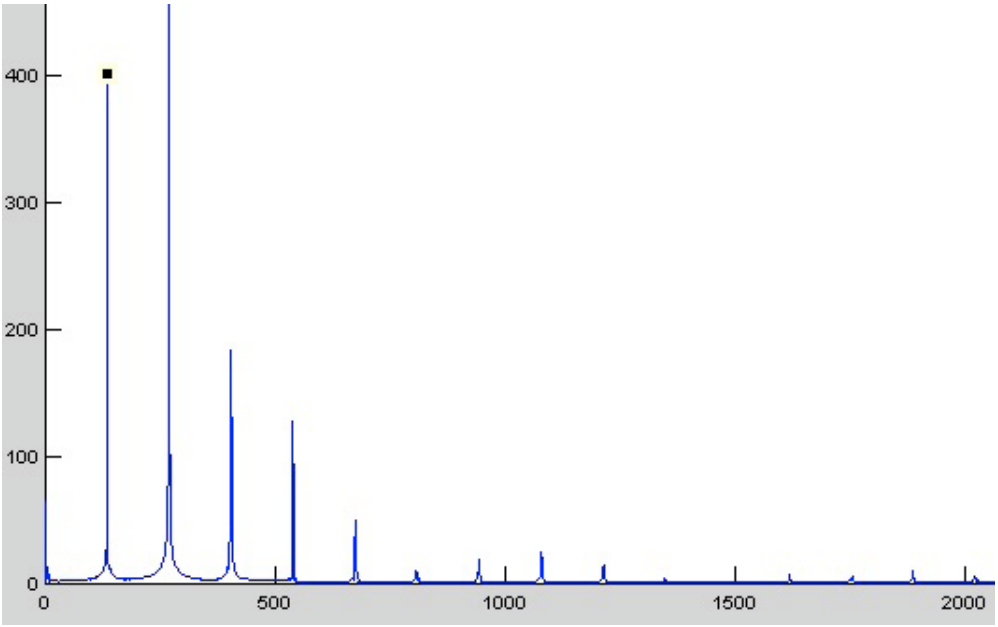
Lab Project Now that we have the FFT, let's start using it. Download the files one.wav through seven.wav, and compute FFT's. These are the tones used in touch-tone phones. What does each represent, and what do you learn about touch-tone encoding?

Lab Project Download the clip from Madonna's version of American Pie. It's sampled at 44100; load it into Matlab. Do the following computations:

```
> z=fft(x, 325240);
> good=ifft(z);
> badreal= ifft(real(z));
> badabs= ifft(abs(z));
```

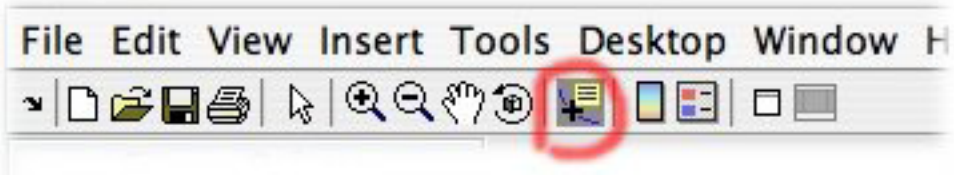
Save these files, then play them. Conclusion?

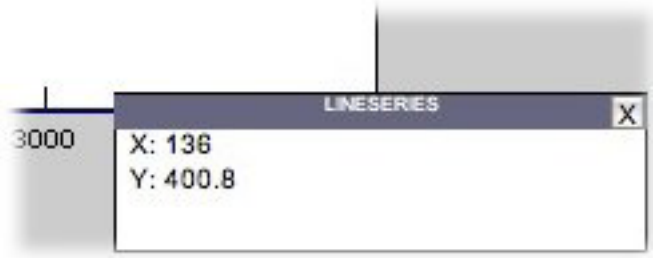
Lab Project Download the clip of a violin playing the note A4. It's sampled at 22050; load it into Matlab. Compute the FFT, and plot the power spectrum. You'll get something like the following:



Find the frequency of each of the first five or six peaks, and show that they are all (approximately) muliples of the lowest frequency. Thus, the violin is playing a fundamental frequency and its harmonics.

Remark: if you can't find the peaks using matlab and trickery, you can do it by hand as follows. Click the circled button below to get yourself into 'point selecting' mode. Then right-click and follow the pop-up menu to 'display characteristics' and select 'in a box'. Then, when you click on a peak, the frequency and height of each peak is shown in a small box.





Lab Project Repeat the process with the linked file of a *didgeridoo,* a musical instrument of the Aboriginal peoples of Australia (sampled at the rather unusual frequency of 16000 Hz). Is it still true that the frequencies are a multiple of the fundamental? This is a tube with one end closed (?)

Lab Project Repeat with the linked file of a chime (sampled at 48000 Hz). The chime doesn't have fixed ends. It vibrates by moving up and down along its length, like a string, but the ends aren't held down. So there's no reason to suppose that the frequencies it can support have to be multiples of a fundamental.

Lecture 2

FOURIER TRANSFORMS

Basics with the FFT

We want to start computing FFT's, to see the kinds of information they give. We'll run several experiments; the idea is to look at FFT's in several different ways, finding out what each tells us. And what will turn out more important -- what each conceals.

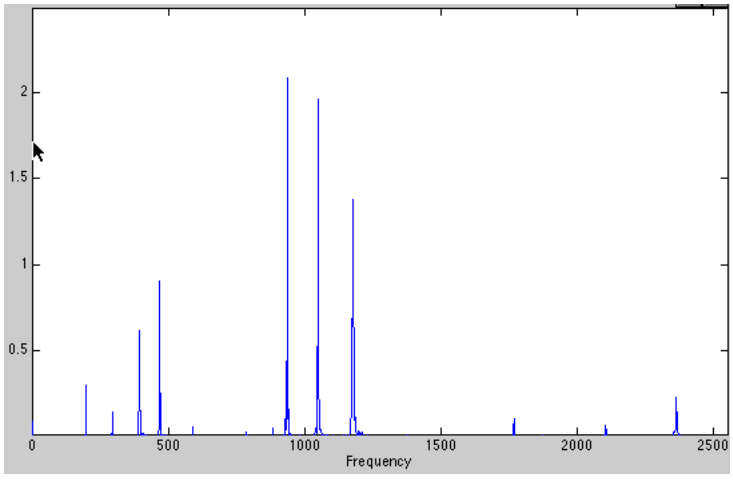
A quick computation before we use Matlab: let's assume we're taking the FFT of some music, from a CD. Every second, we have 44100 numbers, so a file lasting three minutes involves M = 7,938,000 numbers: music can eat up computer memory really fast.

Computers are capable of reading CD's directly, but after that each computer uses its own technique to store the music. Wintel machines use the WAV standard; Macs use AIFF; Sun workstations the AU standard. Compressed music is in either mp3, mp4 or one of the Microsoft compression schemes. By and large don't have to worry about that; the computer programs supplied on the course CD can read all those formats. Matlab can read WAV and AU, so all the sounds provided on the CD are in WAV format.

If we do a FFT based on N=128 points, then we'll be looking at .0029 seconds of the music; we'd find out what happens in those .0029 seconds. This seems a bit limited for a piece of music likely to run maybe three minutes for a pop song. So that's the first issue: why would anyone take N=128, when they could take N=billions and billions?

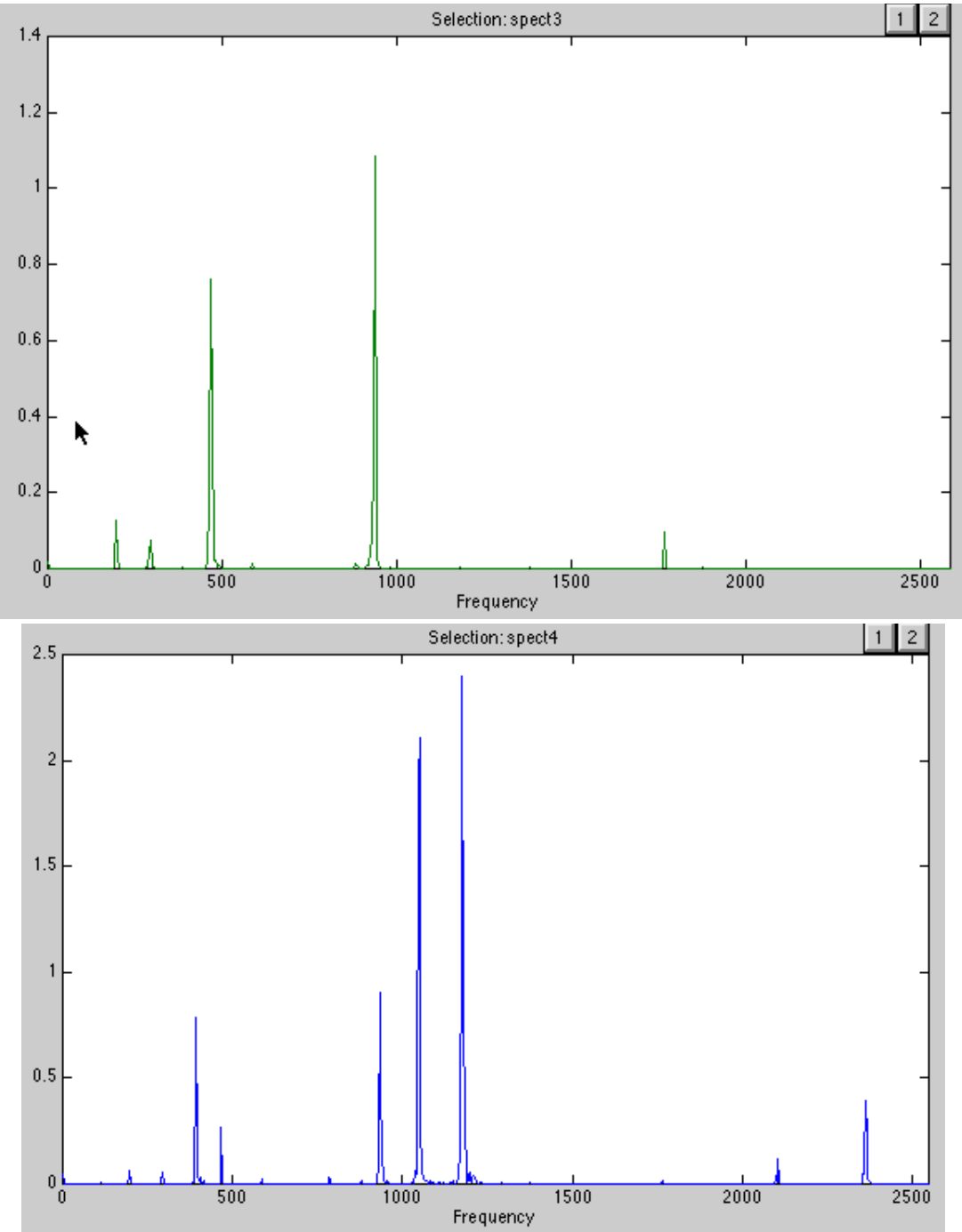
On the flip side (for those of us who grew up with vinyl records, which have flip sides), we might not care what the frequencies are for a whole recording. If we wanted to find when the guitar enters a piece, that's something that'll happen in a fraction of a second. Computing all the frequencies through the whole song blurs time; it conflates what happens over seconds and what happens over fractions of a second.

Here's an example. Play the sound s.wav; it's two notes of a piano, with M about 32600. Below is the spectrum, with N the closest power of 2, N=32768



What we see is that the two notes are intermingled in the spectrum; we can't tell there are two notes, or even which part of the spectrum belongs to which note.

To get at the individual notes we take short chunks of a piece of music. We chop the sound into the left piece (first note) and the right (second note). Play the sounds sl.wav and sr.wav. Here's the spectrum for sl, again computed at maximum -– N=8192, then sr, at max N = 16384



Short-time FFT's are capable of distinguishing events that occur very quickly -- like one note following another on a piano. What we're going to find, though, is that it's a bit of a devil's bargain -- we lose something for everything we gain.

Lab Problem Now for a matlab example: set up a signal of length 4096 (FFT likes powers of two), then some sines of varying frequencies.

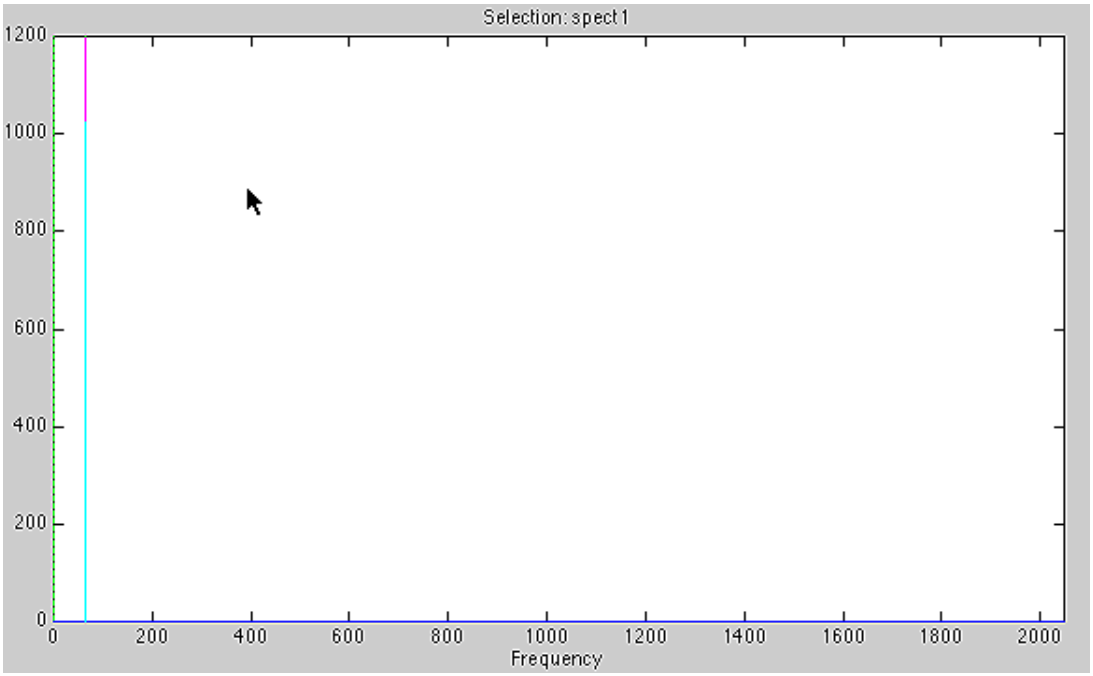
```
x=linspace(0,1, 4096);

%this divides the interval [0,1] into 4096 points; if we
%think of [0,1] as one second of time, then we have sampling at
%4096Hz.

y64=sin(2*pi*64*x);
y71= sin(2*pi*71*x);
sptool
```

Import the three signals into sptool (if you have questions about basic sptool use, you can get the sptool chunk of the maual on the CD, in the manuals folder). When you do your import of y64 and y71, remember to set the sampling rate at 4096.

Now select the y64 signal in the signal part of sptool, then click on the 'create' button in the spectrum part.

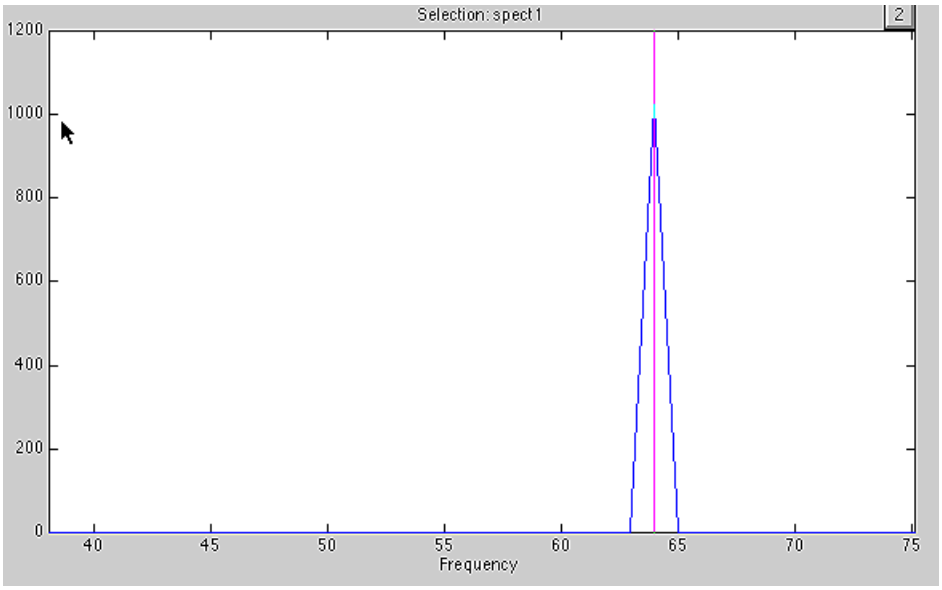
sptool offers a variety of ways to analyze the signal; start by using FFT: select it from the pull down menu. Your next choice is the number of points to analyze; since we have a signal with length 4096, take all 4096 points.

Here's the result, for y64:



You get a strong spectral line, at the far left; you can use sptool capabilities to locate it precisely, by dragging a vertical line over to it, and then reading the frequency from the reporting box at the far rights. It reports a frequency of 64hz.

If you want, you can use the mouse zoom tool to get a close up:

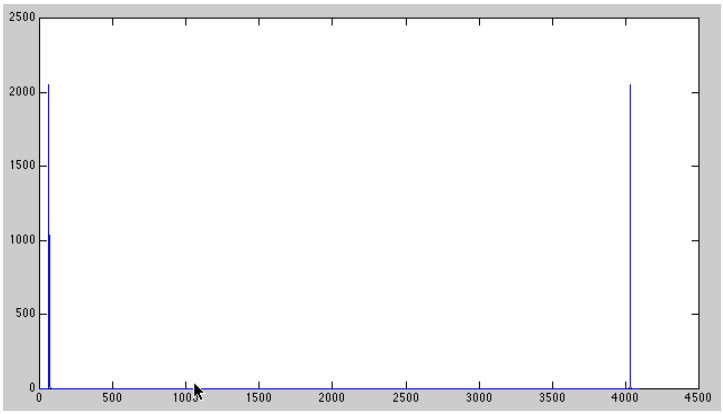


It looks nicer, in a way; certainly you can see the peak at 64 much more clearly. But there's a warning: the little triangle you see isn't really there. After all, the FFT only computes the transform at integer points, and every single point on the slopes of that triangle is at a non-integer point.

sptoool is one way to get at the FFT. You can also do it by hand, using the "fft" command on the Matlab command line. In this case, do

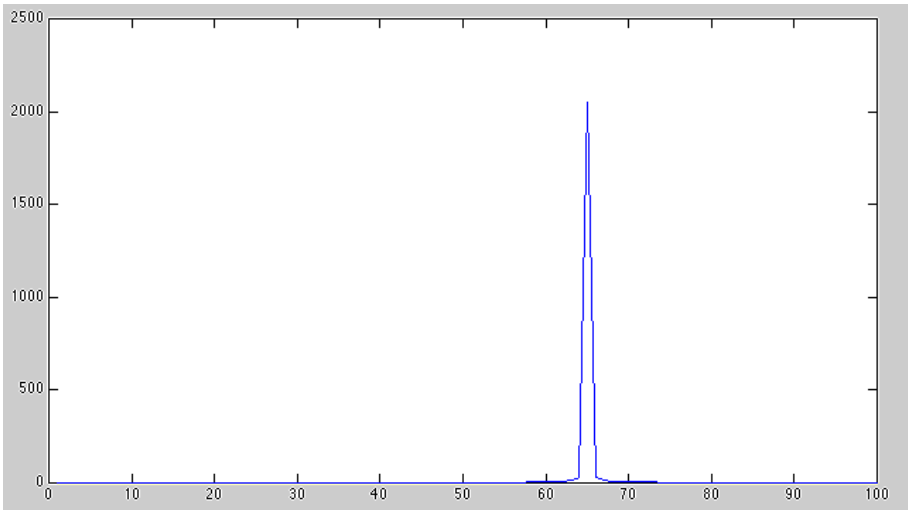
```
z64=fft(y64);
plot(abs(z64))
```

(the 'abs' is there to eliminate the complex numbers ...) Here's the result:



Just what I expected. There's two lines because I know abs(fft(y)) is symmetric when the input signal is pure real. And If you want to zoom, well you can do it here too . . . just plot fewer than all 4096 points:

```
plot(abs(z64(0:100)))
```



Matlab is also capable of giving numbers instead of pictures; just leave off the semicolon. You get

```
abs(z64(55:70))

ans =

   1.0e+03 *

  Columns 1 through 7

    0.0029    0.0033    0.0037    0.0043    0.0051    0.0061
    0.0077

  Columns 8 through 14

    0.0104    0.0156    0.0313    2.0469    0.0328    0.0164
    0.0110

  Columns 15 through 16

    0.0083    0.0067
```

The `1.0e+03 *` means that the answer should be multiplied by a thousand. The numbers make the "spectral peak" idea very vivid: the transform is like `2,046.9` at the peak, and like `3.28` even one unit away from the peak.

But that's still not enough for us: we'll do the computation -- by hand!

We'll do the computation with $y32 = e^{2\pi i(32x)}$ instead of sine. Purists who insist on sine can use the trick $\sin\theta = \left(e^{i\theta} - e^{-i\theta}\right)/2$.

Also, that isn't quite right . . . we're sampling a sine, at points $k/4096$. Let M denote my sampling frequency. Also, since I've got two frequencies, 32 and 37, we'd better use ω for frequency.

OK, ready to go: the signal $y(k) = e^{2\pi i(\omega k)/M}$. Then its N-point FFT is

$$z(n) = \sum_{k=0}^{N-1} e^{-2\pi i(nk)/N} y(k) = \sum_{k=0}^{N-1} e^{-2\pi i(nk)/N} e^{2\pi i(\omega k)/M}$$

$$= \sum_{k=0}^{N-1} e^{2\pi ik[-n/N+\omega/M]} = \sum_{k=0}^{N-1} \gamma^k$$

Now the sum on γ^k depends: if $\gamma = 1$, this sums to N; otherwise it sums to $(1-\gamma^N)/(1-\gamma)$. So let's check it out:

$$\gamma = e^{2\pi i[-n/N+\omega/M]}$$

But we're picking $M = N = 4096$, so all we have is

$$\gamma = e^{2\pi i[-n+\omega]/N}$$

and sure enough, when $n = \omega$, $\gamma = 1$. However, if $n \neq \omega$, then we're in the $(1-\gamma^N)/(1-\gamma)$ situation – in particular the denominator is not zero. Which is very very nice.

But . . . the numerator could be zero. Let's check it out:

$$\gamma^N = \left(e^{2\pi i[-n+\omega]/N}\right)^N = e^{2\pi i[-n+\omega]} = 1$$

We fudged here . . . n is an integer, but in general ω doesn't have to be. But we're looking only at $\omega = 32$ or maybe 37, so $-n+\omega$ is an integer and the exponential is zero and so is the sum.

We've computed, by hand, what the FFT in Matlab showed in a picture. Kinda brings a tear to the eye.

It's nice to know that we reproduce the spectral peak. However, the computation is supposed to show that **`abs(z64(64)) = N = 4096.`** But what Matlab got was **`1.0e+03 * 2.0469 = 2047.5.`** Note twice this is 4093.8. Anyone can get a little numerical inaccuracy, but half the correct answer! What's going on here?

We predicted, based on theoretical computations, that the value of the FFT of something like y64 at the peak should be N. But when we let Matlab actually do the computation, the answer I got is like N/2.

When we did the theory, we computed the FFT of exp(2*pi*i*71*x). But in Matlab, we used sin(2*pi*71*x). Of course the two are related by . . . sin(a) = [exp(i*x)-exp(-i*a)]/[2*i]. The factor of two.

Next, we'll narrow the time resolution: let's take N=128. Since N = 128 in the FFT, the transform exists for indices between 0 and N-1. Since it's a real signal, symmetry guarantees you only get the first 128/2 frequencies.

But -- we sampled the sine at M=4096 samples/second. The Nyquist sampling theorem tells you that the highest frequency you can resolve is half that, you get 4096/2 hz as your top frequency.. And in turn the FFT samples those frequencies at 128 /2 places, giving 4096/128 = 32. The frequencies you get are 128 different frequencies, with a 32 Hz gap between each.

To do the Matlab, you have to type "**`fft(y64, 128)`**" to get the transform at N=128 points. You'll get **`z64=abs(fft(y64, 128));`** and this will be a 1 by 128 matrixx. But, z64(1) really is the transform evaluated at zero (due to the way Matlab indexes vectors, they can't start at zero) and z64(2) isn't the transform at the frequency 2hz; it's the transform at (2-1)(32) hz; z64(2) is the transform at (3-1)(32)=64 hz, and so on all the way up.

Here's the Matlab code:

```
z64=abs(fft(y64, 128));

z64(1:10) =

Columns 1 through 7

0.0015    0.0209   63.9922    0.0375    0.0208    0.0149    0.0117

Columns 8 through 10

0.0098    0.0084    0.0074
```

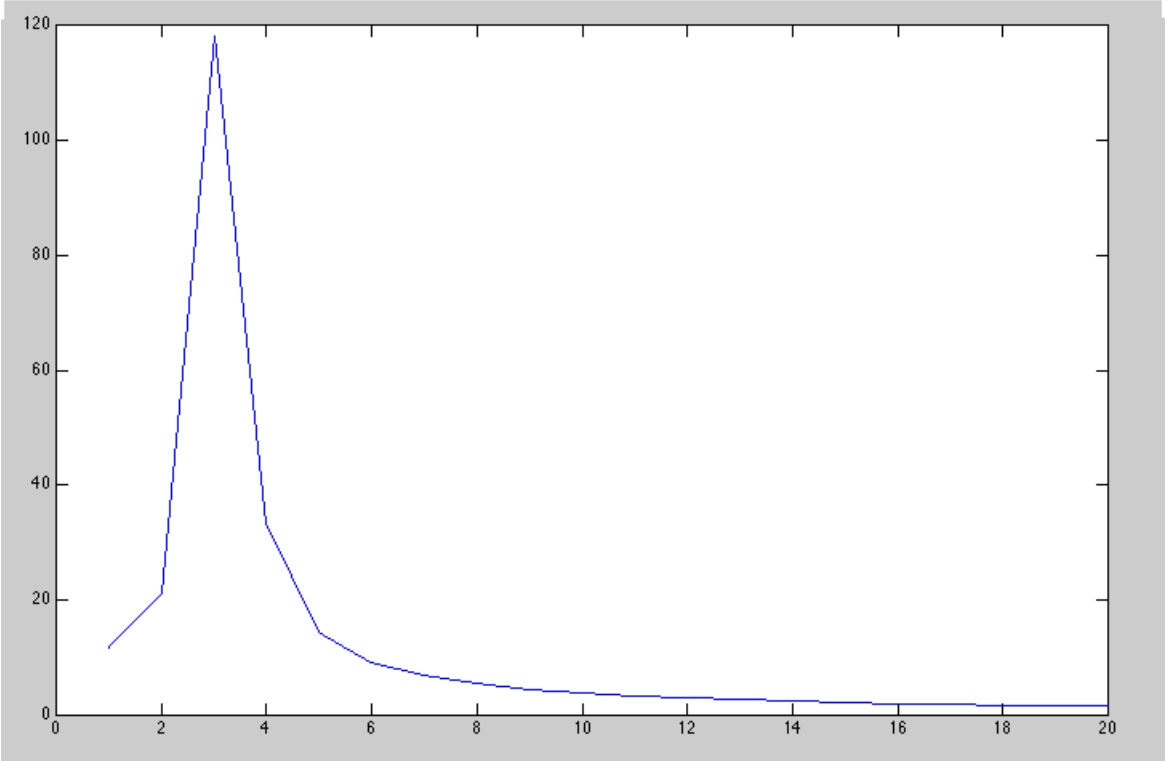
Which matches our expectations. The third entry is the transform evaluated at 64hz, and it ought to return N=128/2 (half because of the sine). It goes without saying we want to compute this by hand, using the definition of the FFT. Why miss a chance for fun? After all, the old computations don't apply, since M and N aren't equal anymore!

Remark We ought to confess right now: there's an actual reason for all this duplication. In the very next example, the results Matlab gives us won't be at all right, and we'll *need* to go to the FFT hand computations to understand why.

Let's go back to the sum on γ^k. This time, $M = 4096,\ N = 128,\ \omega = 64$. Then the argument of the exponential is $2\pi i\,[-n/128 + 64/4096]$, and this can be made zero precisely if $n = 2$. In this case, $\sum \gamma^k = N$, just as before.

For other n, it's just like before; we need the transform to be zero, and the only way that could happen would be for γ^N to to 1. But the argument of γ^N is $2\pi i\,[-n/128 + 64/4096] \cdot 128 = 2\pi i\,[-n + 2]$, which gives an integral multiple of $2\pi i$, hence, $\gamma^N = 1$ and $\sum \gamma^k = 0$. All as before, and, all in agreement with the Matlab results.

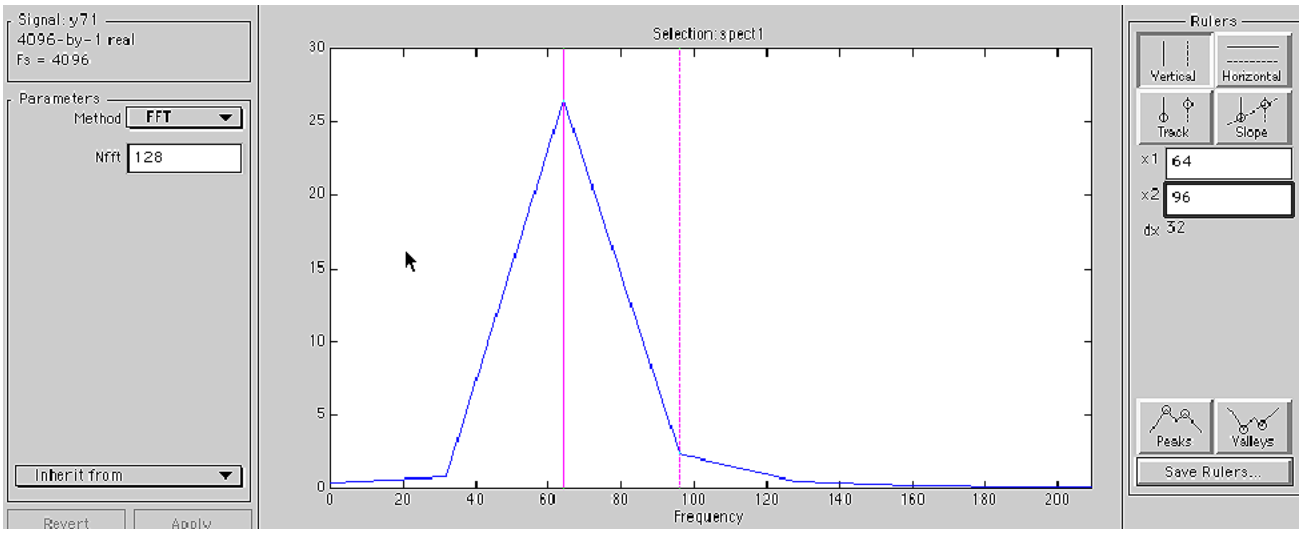
OK. Now the real test -- this time M=4096, N=128, as before, but we'll analyze y71. Here's what the FFT gives:



Here are the numbers:

```
z71(1:7)

ans =

Columns 1 through 7

6.9212   10.1797   58.2163   17.4910    8.1449    5.4748    4.1857
```

We're trying to locate the peak, and where it occurs. We see the peak z71(3) =58.2163, which is the value of the FFT at n=2, which represents the frequency 2*32 = 64. Which can all get a little vague late at night. Fortunately, sptool is good at getting all the indexing right, so let's cross-check using sptool:



Yup: there's a kind of a peak, at 64 (in the report box at right), but it isn't as peaked as z64 was. Also there are non-zero values off to the left and right of the peak, which again is not what happened with z64. The value at the right is at 96hz, that at the left is at 32. Well, yeah of course: 3*32 = 96, 1*32 = 32; these are the frequencies the FFT computes; all values would have to be at 0, 32, 64, 96, etc. But why are they non-zero?

The non-peakinessis clear from the numbers;

```
6.9212   10.1797   58.2163   17.4910    8.1449    5.4748    4.1857
```

The problem here is that the FFT gives me exactly the values at frequencies at 0, 32, 64, 96, 128, . . . 64 is on the list, and when I did the computation, I got the FFT. But 71 is not on the list, and this FFT cannot give me the value at 71.

It seems we get a peak at 64Hz because 64Hz is closest to 71Hz, not because it's the right answer.

How can we explain these results from the formulas?

We left our computation at $(1-\gamma^N)/(1-\gamma)$, where

$$\gamma = e^{2\pi i\ [\omega/M - n/N]}$$

Except this time, $M \neq N$, and, working it out, we get

$$\begin{aligned} \frac{1-\gamma^N}{1-\gamma} &= \frac{1-\{e^{2\pi i\ [\omega/M-n/N]}\}^N}{1-e^{2\pi i\ [\omega/M-n/N]}} \\ &= \frac{1-e^{2\pi i\ [\omega N/M-n]}}{1-e^{2\pi i\ [\omega/M-n/N]}} \end{aligned}$$

This is of the form $(1 - e^a)/(1 - e^b)$, which can be rewitten as

$$\frac{(e^{-a/2} - e^{a/2})}{(e^{-b/2} - e^{b/2})} \frac{e^{a/2}}{e^{b/2}}$$

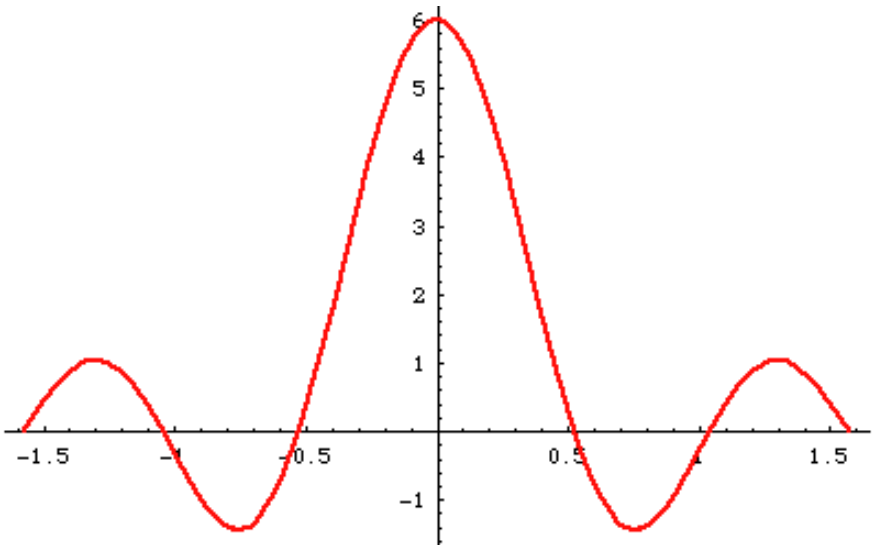
Since $a,\ b$ are imaginary, the difference is a sine . . .

$$\frac{\sin(a/2)}{\sin(b/2)}$$

which isn't quite right, because it ignores the factors $e^{a/2}/e^{b/2}$. But when we take the absolute value, these become one, so, in all, we get

$$\frac{\sin(\pi N\,[\,\omega/M - n/N])}{\sin(\pi\,[\,\omega/M - n/N])}$$

It's worth seeing what this function looks like. Below, the graph of $\sin(10\theta)/\sin(\theta)$.



Note the strong peak at $\theta = 0$, and a falling off from there. The width of the peak can be roughly measured by the distance between the first two zeroes of the function; that is, the first two zeroes of $\sin(5\theta)$, which will be at $5\theta = +\pi,\ -\pi$. The size of the central portion then is roughly $2\pi/5$.

Applying this to

$$z37(n) \ = \ \frac{\sin(\pi N\,[\,\omega/M - n/N])}{\sin(\pi\,[\,\omega/M - n/N])}$$

we expect the function to peak when $\omega/M - n/N = 0$, that is, when $n = (N/M)\omega$. Recall that $M = 4096,\ N = 128,\ \omega = 37$, so that this occurs when $n = 37/32$. That is, never: we only have $n = 0,\ 1,\ , \ldots,\ 127$. The discrete function $z37(n)$ will peak for the n closest to $37/32$, that is, for $n = 1$.

So why does our FFT graph show a peak at 32?

Because n isn't the frequency; it's the number indexing the chunk of frequencies that the FFT can compute. Once again . . you have frequencies from 0 to 4096, divided into 128 chunks, each of size 32, so that the k^{th} chunk represents frequencies between $32k$ and $32(k+1)$. Our function $z37$ peaks somewhere between 32hz and 64hz; that's all we can say.

Going back to the numbers . . .

```
z71(1:7)

ans =

  Columns 1 through 7

    6.9212   10.1797   58.2163   17.4910    8.1449    5.4748  4.1857
```

It's interesting to see what the peak is . . . **`z71(3)= 58.2163.`** Whoa.

This is just weird, especially in comparison with z64. There are three weirdnesses:

Weird I) The peak is at n=2, which represents the frequency 64hz. But the real frequency in the signal y71 is at 71hz. The fft z71 is giving a false value for the peak frequency.

Weird II) Again by comparison with the y64, z64 example: the value of z64 at the peak was around 128/2 = N/2 = 64 (that factor of 2). Here . . . 58?? What's *that* about?

Weird III) There are values of z71, besides the value at the peak. Values, for example, to the left and right of the peak. These represent false frequency readings, reported at 32hz, 96 hz, and a bunch others. That didn't happen for z64! For z64, there was a large value at the peak frequency, then zeroes everywhere else. So, what is the deal? Why does z71 give false frequencies?

In stead of an immediate answer, right away, we want to explain where the numbers came from. So that we could at least reproduce the false values and false frequencies, independent of Matlab. Later will come a real explanation.

The explanation is, it's due to the Dirichlet kernel. The Dirichlet kernel, everyone will remember, is what we get when I take an exponential, sampled at frequency M, and then take the FFT at N points. We did that a few pages back, and got the formula as

$$Dirichlet \;=\; \frac{\sin(\pi N\,[\,\omega/M - n/N])}{\sin(\pi\,[\,\omega/M - n/N])}$$

Let's check this. First, run Matlab:

```
x=linspace(0,1,4096);
yy1=exp(2*pi*i*71*x);
z71=abs(fft(y71,128));
```

Now take an exponential of frequency 71, and compute its FFT, with N=128.

Now check some of the values. There's . . umm, 128 of them, with frequency bins k = 0, 1, 2, . . . , 127. But they represent frequencies k(M/N), that is, 0, 32, 64, 96, and so on.

With y having a frequency of 71hz, the closest bin is at k=2. Hence, of the 128 different values of z, only the first ten or so are close to k=2. We won't bother with the rest:

```
z71(1:20)
ans =
Columns 1 through 7
        21.2459  118.1130   33.1784   14.5500    9.3218    6.8607
Columns 8 through 14
        5.4303    4.4957    3.8375    3.3492    2.9727    2.6738    2.4308
 Columns 15 through 20
        2.2295    2.0602    1.9158    1.7914    1.6831    1.5880
```

Now print out the values of the Dirichlet kernel, *w*=71, M=4096, N=128. Here goes:

```
a=linspace(1,20,20);
c=ones(size(a));
b=abs(sin(pi*128*((71/4096)*c-(1/128)*a))./sin(pi*((71/4096)*c-(1/128)*a)))
b(1:20)

b=
Columns 2 through 7
21.2114  118.1606   33.0868   14.5155    9.3007    6.8455    5.4184
Columns 8 through 14
4.4860    3.8293    3.3420    2.9664    2.6681    2.4257    2.2248
Columns 15 through 20
2.0559    1.9118    1.7876    1.6795    1.5847    1.5008
```

Quite close! Now we can answer some of the questions.

WEIRD I) Why is the peak at n=2, i.e., at the frequency 64hz, instead of at 71hz?
ANSWER: We never computed the frequency at 71, so how can we know what it is? The only thing computed was the frequency at multiples of 32. And 71 isn't. (by the way, later in this lecture we'll show a way to compute the fft at 71, so we aren't giving up). But it isn't that the FFT gave me the "wrong" peak; I simply asked to find the FFT at the wrong frequencies.

The problem here is with N = 128. We decided we were going to analyze 128 samples, or .0029 seconds of this signal. We know very precisely(to within .0029 sec), the *when* of this signal .

It isn't the "when" that's a problem, it's 'when this signal is happening, what frequencies are part of it?' I can only answer that . . . for N=128 points, equally spaced by M/N = 4096/128 = 32hz.

There's a basic fuzziness about the frequencies. Now this is worth going into more, because when we started with these signals, we took M=N=4096. Then the accuracy of the frequency was M/N = 1. And you can see this for yourself. Go to Matlab, make y71 and its fft z71, and see that z71 peaks at 71, and goes to zero at 70 and 72.

But there's a trade-off, to get this great frequency resolution, we'll have to take M = 4096 points, which is 32 times as long a signal as for M=128 points. So, with M=4096 I know the frequencies that occur know 32 times better than I knew the frequecies with M=128, I completely lose the **when** it happens . . in fact, for M=4096, I know the 'when' 32 times worse than for M=128.

There's a nice way to write this: (frequency resolution)*(time resolution) = constant.

This is called the uncertainty principle. You cannot have both good frequency and good time accuracy. If you know physics, there's a law called the Heisenberg uncertainty principle, which is the same thing. In physics.

WEIRD II) For z64, the value at the peak frequency is N/2 = 64. But for z71, the value at the peak is 58. Why isn't it also N/2?

Answer: The Dirichlet kernel tells us. The fft of signals like z71 is (up to absolute value)

$$Dirichlet \;=\; \frac{\sin(\pi N\,[\,\omega/M - n/N])}{\sin(\pi\,[\,\omega/M - n/N])}$$

And this peaks when the argument of sine is zero. Solving for n, that'd be n = (128/4096)71 = 2.21875. But we're computing z71(2), slighlty off from the peak. Hence the value of z71 is slightly smaller than N=128. Bingo. At least a sort of bingo; at least we can predict what the value of z71 comes out to be. We sort of "understand" it, in the sense that we can compute it.

WEIRD III) is, the values of z71 away from the peak aren't zero enough. For z64, they are totally zero away from the peak.

Answer: For z71, the values are given by the Dirichlet kernel. Dirichlet falls off, but it isn't zero. The

next value along, from the 118, is about 33. That's a fall-off; about -11db off from the peak.

Nothing is 'weird;" the values of z71 are all perfectly predictable. They're what we get from the Dirichlet kernel. We understand that the signal y71 has frequency of 71hz, and with N=128 we can only find the fft perfectly accurately for 64hz, 96hz, etc. We're good with this inaccuracy business.

This is different from **understanding** it. Here's why. For z64 none of this happens; you ask the fft what frequencies are in the signal y64, and it tells you: 64hz. You ask z71, and it says, well, I don't know exactly, but the closest is 64hz. But FFT won't tell you better than that.

Then too, when we ask what **other** frequencies are in the signals, z64 and z71 again give different kinds of answers. Ask FFT, what's in the signal y64. FFT says "a peak at 64hz and zero everywhere else". But z71 doesn't tell me that. It tells me, "oh, a peak at 64hz, also there's 21 units of the 32hz, and about 33 units of the 96 hz and. . . . " and all that is WRONG. z71, it seems, makes up all sorts of frequencies that were never even in the signal y71.

It's no good saying 'well, that's because the Dirichlet kernel isn't zero at those frequencies.' z64 don't have no Dirichlet kernel; why does z71? And by the way, why does this issue totaly not arise when N=4096?

The answer turns out to be fairly tricky. The FFT is a tool for analyzing periodic signals. Now that's fine for y64 or y71, when N=4096. Check it out(remembering that x(k) starts at k=1 and ends at k=4097):

y64(1)=sin(2*pi*64*x(1)) = sin(2*pi*64*0) =0
y64(4097)=sin(2*pi*64*x(4097)) = sin(2*pi*64*1) =0

And we can do the same computation with y71 at N=4096. It's still periodic.

The difference is when we try checking periodicity for N=128. Try it first for y64:

y64(129)=sin(2*pi*64*x(129)) = sin(2*pi*64*128/4096) = sin(2*pi*64/32) = 0;

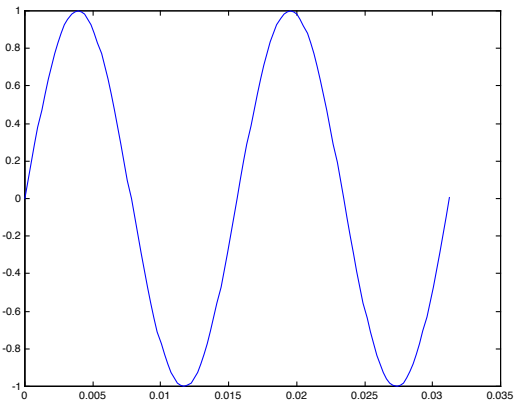
as above, y64 is periodic on N=128 points.

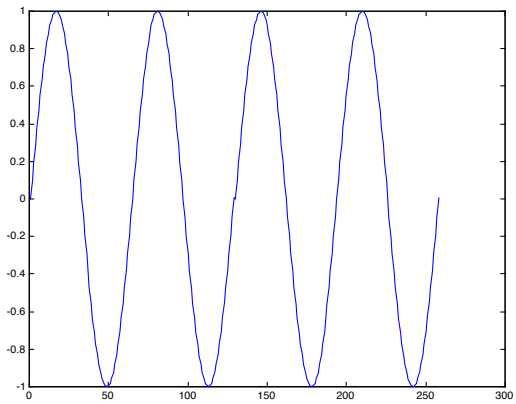
BUT, try it for y71 . . .
y71(129)=sin(2*pi*71*x(129)) = sin(2*pi*71*128/4096) = sin(2*pi*71/32) = .9807 . . .

and this is not zero. y71 loses it's periodicity when you restrict it to 128 points. In fact, yB will lose periodicity unless . . . wait for it . . B =32, 64, 96, . . . all those frequencies that the FFT computes just fine.

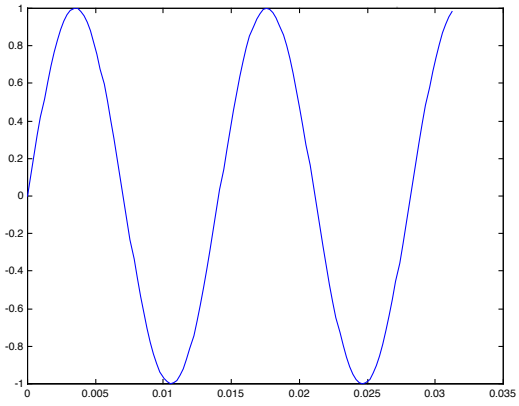
This is all very nice to know, but . . . . how does it explain Dirichlet kernels? Think of the FFT another way: it computes the Fourier transform of the periodic extension of y64 and y71. So let's see what those periodic extensions look like.

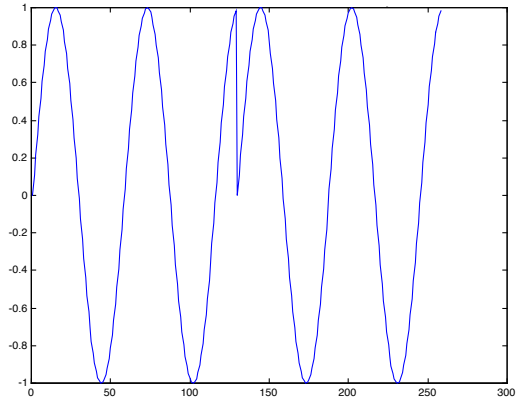
First, y64(k) for k=1, . . . 129:





The graph on the left ends where it started, so when we extend periodically, on the right, the two join up. Compare this with y71:





This is exactly what we got when we did the computation: y71 does not end where it started. Now when we extend periodically by copying and joining graphs, we get the graph on the right, a discontinuity. But the theory of Fourier transforms tells me what happens at a discontinuity: the Fourier transform decays like 1/x. More precisely, let's take the Fourier transform of a very basic discontinuous function: the function that's 1 from -a to a, and zero otherwise:

$$\int_{-\infty}^{\infty} f(x)e^{-2\pi ix\xi}\,dx = \int_{-a}^{a} \mathbf{1}e^{-2\pi ix\xi}\,dx$$

$$= \left[-\frac{1}{2\pi i\xi}e^{-2\pi ix\xi}\right]_{x=-a}^{x=a} = -\frac{1}{2\pi i\xi}\left[e^{-2\pi ia\xi} - e^{-2\pi i(-a)\xi}\right]$$

$$= \frac{\sin(2\pi a\xi)}{\pi\xi}$$

The continuous version of the Dirichlet kernel . . .

Let's summarize: y71 is not a periodic function on 128 points. When you try to periodize it, you get a discontinuity. The transform of a discontinuous function involves a Dirichlet kernel. The Dirichlet kernel doesn't peak at a single point; it has all sorts of non-zero values. And, finally, those non-zero values make the FFT of y71 seem to have all sorts of strange frequency components.

This whole phenomenon is so well-known (in a'watch out for this' kind of way) that it's been given a name: "leakage". The idea is that the 71hz frequency won't fit into 64hz, and it leaks out into the other frequencies.

Cute term . . but . . why? Why does the FFT of a discontinuous function involve a Dirichlet kernel? Why does the Fourier transform of a discontinuous function involve all those weird not-right frequencies?

Because discontinuity is a high-frequency phenomenon. Let's try thinking about discontinuity, and sudden change. I'll measure 'change' by computing the derivative.

Thought experiment: if y = sin(2*pi*f*x), what is the maximum value of the derivative of y? Well , 2*pi*f. So the fastest change that can happen is proportional to the frequency. So we say, "fast change = high frequency."

In the Dirichlet case, we have a discontinuity; the derivative is extremely large, hence the frequencies involved in representing the change must be large as well. That's all there is to this business.

After all this, it is so tempting to day d'uh! Of course a discontinuity gives us a lot of high frequency junk. I knew that! But really I've done a lot more; the Dirichlet kernel makes it all precise and predictable. And it serves as awarning: be careful using the FFT; it may not be telling you what you think.

Still and all, this "uncertainty" is pathetic! The FFT is an exact copy of the original signal! Just take the IFFT, and you'll see your original 71hz sine. So, the information about the correct frequencies is in the FFT – we just can't seem to get it out, for some reason.

Where the information is, is in the phases we've seen/heard that from the lab on nonlinearities. So how do we extract the info?

Let's say we have a sampling frequency of f_s. The FFT is a sample of the DFT; the DFT exists at all values of θ, $0 \leq \theta < f_s$. The FFT samples these frequencies at $\theta[k] = f_s k/N$ for $0 \leq k < N$. Hence our problem. But that's only a very few of the frequencies in the DFT.

Of course the FFT ignores those other frequencies because the FFT is very fast to compute by ignoring those other frequencies. Doing the intermediate frequencies would slow down the computations, not to mention that we couldn't use any of the built-in Matlab functions like FFT to do our computations.

Fortunately, there's a very cool gimmick called the chirp transform. It's just about as fast as the FFT, and allows us to compute those intermediate frequencies.

Chirp starts by assuming you have some basic frequency θ_0 that you want to focus in on. It also assumes you have some resolution $\Delta\theta$ that you want to achieve (presumably better than $f_s k/N$) And that you have some range of frequencies, starting with θ_0, that you want to know. So you'll be

And that you have some range of frequencies, starting with θ_0, that you want to know. So you'll be computing the Fourier transform at $\theta[k] = \theta_0 + j\Delta\theta$ for $j = 0,\ 1,\ \ldots, K$. You want

$$\begin{aligned} z(\theta_0 + j\Delta\theta) &= \sum_{k=0}^{N-1} e^{-2\pi i(\theta_0 + jk\Delta\theta)} y(k) \\ &= e^{-2\pi i\theta_0} \sum_{k=0}^{N-1} e^{-2\pi ijk\Delta\theta} y(k) \end{aligned}$$

But $2jk = j^2 + k^2 - (k-j)^2$. Hence,

$$\begin{aligned} e^{-2\pi ijk\Delta\theta} &= e^{-2\pi i\frac{1}{2}\left[j^2 + k^2 - (k-j)^2\right]\Delta\theta} \\ &= e^{2\pi i\frac{1}{2}(k-j)^2\Delta\theta} e^{-2\pi i\frac{1}{2}k^2\Delta\theta} e^{-2\pi i\frac{1}{2}j^2\Delta\theta} \end{aligned}$$

Now plug back into the sum, to get

$$\begin{aligned} z(\theta_0 + j\Delta\theta) &= e^{-2\pi i\theta_0} \sum_{k=0}^{N-1} e^{2\pi i\frac{1}{2}(k-j)^2\Delta\theta} e^{-2\pi i\frac{1}{2}k^2\Delta\theta} e^{-2\pi i\frac{1}{2}j^2\Delta\theta} y(k) \\ &= e^{-2\pi i\theta_0} e^{-2\pi i\frac{1}{2}j^2\Delta\theta} \sum_{k=0}^{N-1} e^{2\pi i\frac{1}{2}(k-j)^2\Delta\theta} e^{-2\pi i\frac{1}{2}k^2\Delta\theta} y(k) \end{aligned}$$

. If I define

$$\begin{aligned} a(k-j) &= e^{2\pi i\frac{1}{2}(k-j)^2\Delta\theta} \\ b(k) &= e^{-2\pi i\frac{1}{2}k^2\Delta\theta} y(k) \end{aligned}$$

then what we have is

$$\begin{aligned} z(\theta_0 + j\Delta\theta) &= e^{-2\pi i\theta_0} e^{-2\pi i\frac{1}{2}j^2\Delta\theta} \sum_{k=0}^{N-1} a(k-j)b(k) \\ &= e^{-2\pi i\theta_0} e^{-2\pi i\frac{1}{2}j^2\Delta\theta} a \star b(j) \end{aligned}$$

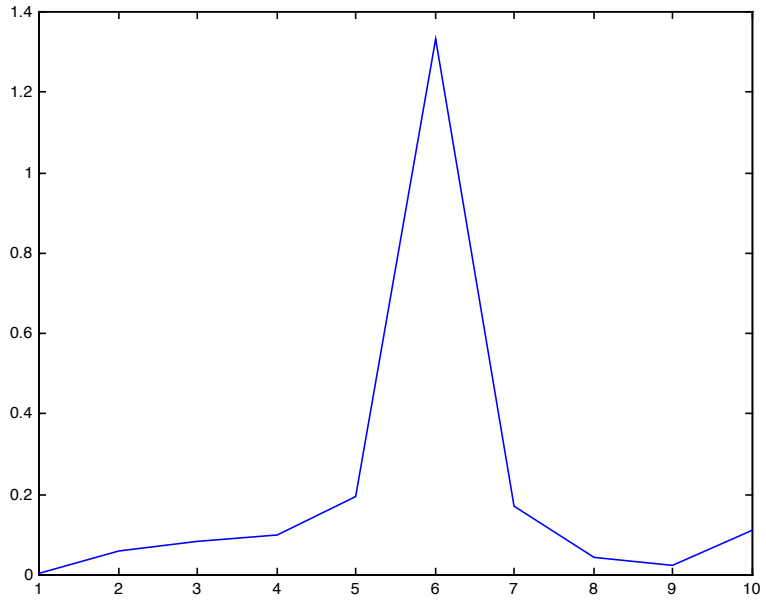
Up to a phase factor, this is a convolution. But convolutions and FFT's are easily related; $a \star b =$ IFFT [FFT [a]] $\cdot$ [FFT [b]].

Now to implement it in Matlab: you write an M-file, and save it as chirpf.m (you have to be careful; chirp.m is an already-defined, different M-file). Here's the code; we've taken it from the book of Boaz Porat, A Course In Digital Signal Processing

```
function X= chirpf(x, theta0,dtheta,K);
%Synopsis: X= chirpf(x, theta0,dtheta,K).
%Computes the chip Fourier transform on a frequency
%interval.
%Input parameters:
%x : the input vector
%theta0 : initial frequency in radians
```

```
%dtheta : frequency increment in radians
%K : number of points on the frequency axis
%X : ouput, the chirp Fourier transform of x

N=length(x); x=reshape(x,1,N);n=0:N-1;
g=x.*exp(-j*(0.5*dtheta*n+theta0).*n);
L=1; while (L<N+K-1),L=2*L;end
g=[g,zeros(1, L-N)];
h=[exp(j*0.5*dtheta*(0:K-1).^2),...
exp(j*0.5*dtheta*(-L+K:-1).^2)];
X=ifft(fft(g).*fft(h));
X = X(1:K).*exp(-j*0.5*dtheta*(0:K-1).^2);
```

Below, what you get with theta0=96, dtheta = 1 and K = 10, applied to the signal y101:



Let's see, what's going on in this picture? We're analyzing the spectrum of y101, starting at 96hz and continuing on to 107hz, in steps of one. So we ought to get the correct values of the frequencies at 96, 97, 98, 99, 100, 101, etc. And check it out, the sixth value along shows a sharp peak: the frequency at 101. (note you shouldn't be trying 96 + 6 = 102; 1 on the horizontal axis corresponds to 96hz; because of the way Matlab labels matrices, the first element always has index one, not zero)

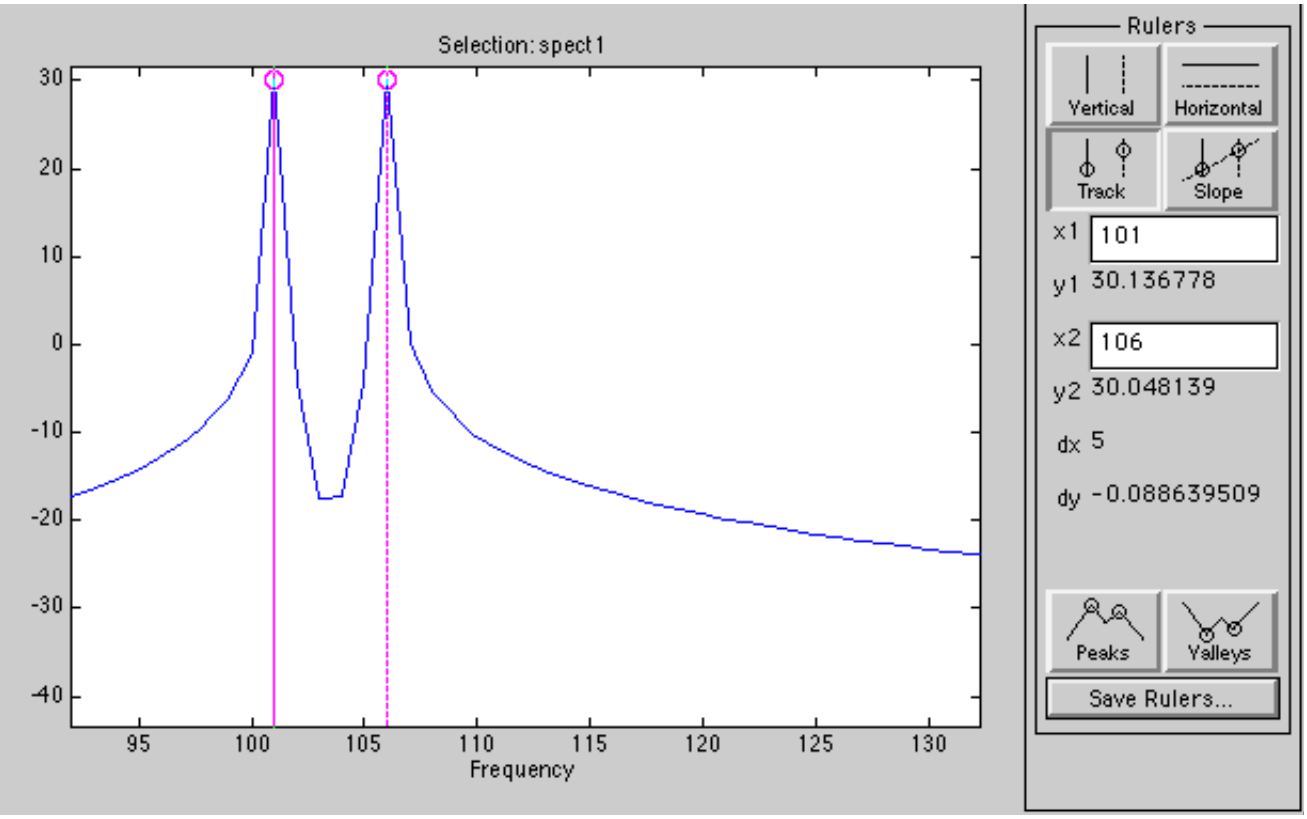
Chirp gives me the correct values, even when the FFT is flaking out.

Before going on, we need to confess: Matlab has a built-in chirp transform. It's part of a more general program, czt, the "chirp z-transform'. There's a detailed discussion in the manual for the signal processing toolbox; I put czt manual pages in the MANUALS folder on the CDROM. It also gives a reference to a book that discusses the chirp theory, in case you don't like our discussion.

For the other version, type '**`help czt`**' when you're in Matlab; here's what you get:

```
CZT  Chirp z-transform.
G = CZT(X, M, W, A) is the M-element z-transform of sequence X,
where M, W and A are scalars which specify the contour in the z-
plane on which the z-transform is computed.  M is the length of the
transform, W is the complex ratio between points on the contour,
and A is the complex starting point.  More explicitly, the contour
in the z-plane (a spiral or "chirp" contour) is described by
z = A * W.^(-(0:M-1))

The parameters M, W, and A are optional; their default values are
M = length(X), W = exp(-j*2*pi/M), and A = 1.  These defaults cause
CZT to return the z-transform of X at equally spaced points around
the unit circle, equivalent to FFT(X).

If X is a matrix, the chirp z-transform operation is applied to
each column.

See also FFT, FREQZ.
```

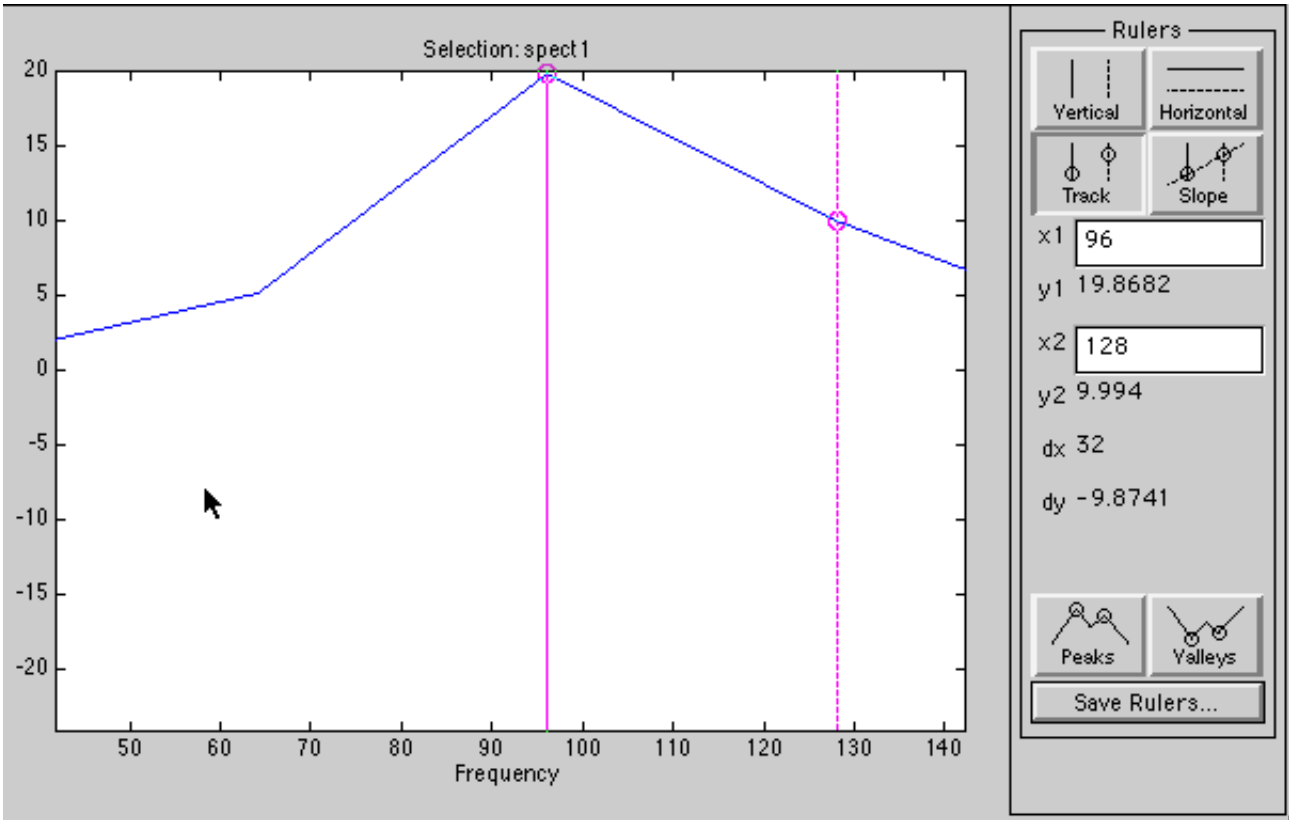
We'll leave behind chirp and friends, to give another perspective on the poor resolution of the FFT and the uncertainty principle; we want to see what the FFT does when it gets two frequencies. Let `y101_106 = sin(2*pi*101*x)+ sin(2*pi*106*x)`. Here it is, first at N=4096



You see the two separate peaks, at 101hz and 106hz, very clearly. No surprise: we already know that if N=4096, the resolution M/N of the FFT is 1hz. And our two frequencies, 101 and 106hz, are separated by substantially more than 1hz. But: now try the analysis again at, but with

```
y96_106 = sin(2*pi*96*x)+ sin(2*pi*106*x)
```

at N=128.



That is not good. This transform doesn't have the two peaks it should, just one, and it's at 96hz. The peaks issue is same-old, same-old: k=96 is one of the frequencies that the FFT can represent exactly at M=4096, N=128, but k=106 is not representable exactly. Besides: the resolution is M/N = 4096/128 = 32hz, and roughly what this means is that any signals with frequencies closer than 32hz will be blurred together.

The Dirichlet kernel gives a way of thinking about the way the FFT blurs the spectrum of y96 and y106 together.

We expect the center portion of the Dirichlet kernel

$$\frac{\sin(\pi N\,[\,\omega/M - n/N])}{\sin(\pi\,[\,\omega/M - n/N])}$$

to extend all the way between the first zero to the left, when the argument of sine is $-\pi$, to the first zero on the right, at $+\pi$. Any frequencies ω in that range will have a FFT whose central value is blurred together with all the other frequencies. The blurring thus holds for

$$
\begin{aligned}
-\pi \ &\le \pi N\,[\,\omega/M - n/N\,] \ \le \ \pi \\
-\frac{1}{N} \ &\le \ [\,\omega/M - n/N\,] \ \le \ \frac{1}{N} \\
\frac{n}{N} - \frac{1}{N} \ &\le \ [\,\omega/M\,] \ \le \ \frac{n}{N} + \frac{1}{N} \\
\frac{M}{N}(n-1) \ &\le \omega \ \le \ \frac{M}{N}(n+1)
\end{aligned}
$$

In our case, $M/N = 32,\ n = 2$. Thus, frequencies ω with $32 \le \omega \le 96$ all get blurred together under the central peak at 64.

Obviously, this is an exaggeration; the Dirichlet kernels can overlap without merging. But it does explain, in a general overview kind of way, the blur of the uncertainty principle.

(By the way, to test how much an exaggeration this is, try graphing sums of Dirichlet kernels, shifted a bit, to see how far you have to shift before truly merging).

There's more info in the Dirichlet kernel; it tells us how fast the FFT has to fall away, after it hits the peak frequency at ω. Let's try and find how tall the peaks of the Dirichlet kernel actually are.

Try this. The Dirichlet kernel $\sin(\pi N\theta)/\sin(\pi\theta)$ has its extremes when the derivative is zero, which, a quick check tells you, is at θ satisfying the equation $\tan(\pi N\theta) = N\tan(\pi\theta)$. It isn't hard to see that the zeroes of tan are solutions, but where are the others? Damn, I can't solve that equation.

A crude way to estimate the peaks is that between every two zeroes of a continuous function f is a peak (known as Rolle's Theorem!).

So let's us find the zeroes of the Dirichlet kernel, and *approximate* the peaks as being half-way in-between.

So. The Dirichlet kernel $\sin(\pi N\theta)/\sin(\pi\theta)$ iz zero when the argument of the sin in the numerator is a multiple of π: $\pi N\theta \ = \ \ldots \ , \ -2\pi, \ -\pi, \ 0, \ \pi, \ 2\pi, \ \ldots$. Now zero is a bit of a cheat, because in fact

$$
\lim_{\theta\to 0} \frac{\sin(\pi N\theta)}{\sin(\pi\theta)} \ = \ N
$$

and this is a maximum, as we saw from the picture. So, to look for the next extreme value, we loot for the next two zeroes, which are at $\pi N\theta \ = \ 1\pi, \ 2\pi$, or $\theta \ = \ 1/N, \ 2/N$. Half-way in-betwen is $\theta \ = \ 3/2N$, and the value of the Dirichlet kernel there is $\sin(\pi N(3/2N))/\sin(\pi(3/2N))$ $= -1/\sin(3\pi/2N)$. We expect N to be large, so we expect θ to be small, hence we can use the approximation $\sin\theta \approx \theta$, and, all in all, we expect:

The value of the Dirichlet kernel at its first maximum is N, and at it's second extreme, is $-2N/3\pi$. Similar results hold at the further extremes.

This, we can test. When we computed our Dirichlet kernel, we got values 21.2459 118.1130 33.1784. The ratio of these peaks to the (unseen) peak N=128 are .166, .923, and .259. These should be compared to our approximations, .212, 1, .212. The values aren't too badly off.

So that's it! The FFT is a nice tool for finding the frequencies in a signal, but it comes with a price: uncertainty, leakage and blurring.

One "practical" place this arises is in the construction of vocoders. The point of a vocoder is to shift the frequencies in a song. Play Shakira singing Estoy Aqui, estoy.wav song. That's the way it should sound. If I apply a vocoder to it, Iget estoy_deep.wav and in the other direction, estoy_high.wav. The vocoder has made the singer appear to sing in a higher pitch.

In the labs, you can try to accomplish this using the fft and changing frequencies. But because of the uncertainty principle, the frequencies inbetween the gaps don't get changed. Building a good vocoder involves finding the correct frequencies inbetween the gaps. Which is where chirp and friends come in!

Lecture 2

The Fast Fourier Transform

Power Spectra

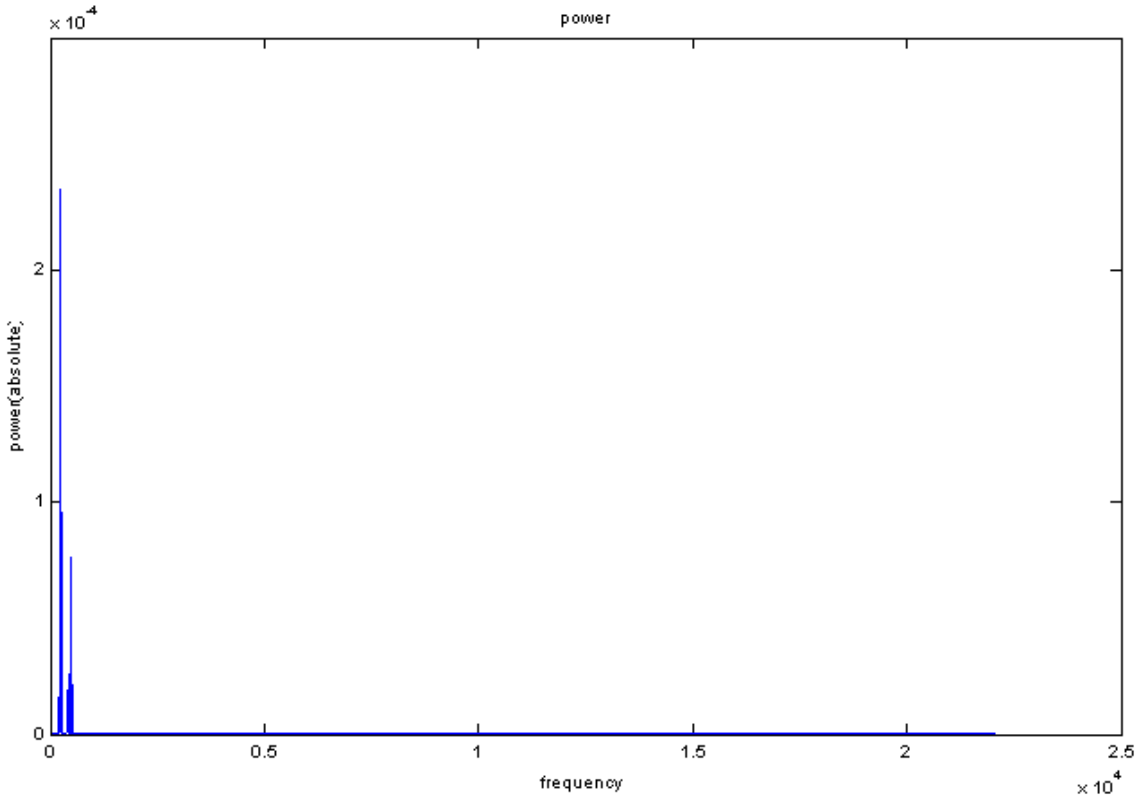
If we want to actually compress music, we have to decide how much to keep and how much to discard. The words we're using – 'how much' already ask us to measure, to attach a number to the music. In this course, we've been suggesting that vector theory is a good guide to understanding signals, and in vector theory, we measure the size of a vector a by its length, $\|a\|$. If $a = (a_1,\ a_2,\ \ldots,\ a_n)$, then $\|a\| = \sqrt{|a_1|^2 + |a_2|^2 + \cdots + |a_n|^2}$.

There's another reason to use this kind of measure: if we have a signal $s = A\sin(2\pi\omega_1 x) + B\sin(2\pi\omega_2 x)$, then basic physics tells us that the *power* carried by this wave is $|A|^2 + |B|^2$. Because of this, if we have a signal $s(n)$ and we take the FFT, $\hat{s}(n)$, the numbers $|\hat{s}(n)|^2$ are said to make up the *power spectrum* of the signal. The total power is then $\sum |\hat{s}(n)|^2$.

One approach to compression is to take the FFT of a signal, and then keep a fixed percentage of the power in the signal – say, 80% or 99%. We can do this fairly easily in Matlab:

```
%read in the signal sezen_44.wav -- a pure vocal, sampled at 44100Hz
>>x=wavread('sezen_44.wav');
% compute the power spectrum, telling the FFT to use length(x) as the number of
%points in the FFT, and to use the sampling frequency 44100Hz.
>>[P,f] = periodogram(x,[],length(x),44100);
%the function returns a vector of frequencies at which the FFT was computed, f,
%and the power at each frequency, P.
```

Here's what it looks like, plotted all the way to the Nyquist limit, 22050Hz.



Notice that the power is concentrated in the lower frequencies. But, there's a lot more of the higher frequencies -- is it possible that the two balance out?

Maybe the simplest way to check is to compute the total power less than a given frequency. Let's start: first, note that we have a lot of frequencies: the range from zero to 22050 has been divided into length(x)/2 frequencies. For example:

```
>f(1:10)

ans =

         0
    0.2468
    0.4936
    0.7404
    0.9872
    1.2339
    1.4807
    1.7275
    1.9743
    2.2211
```

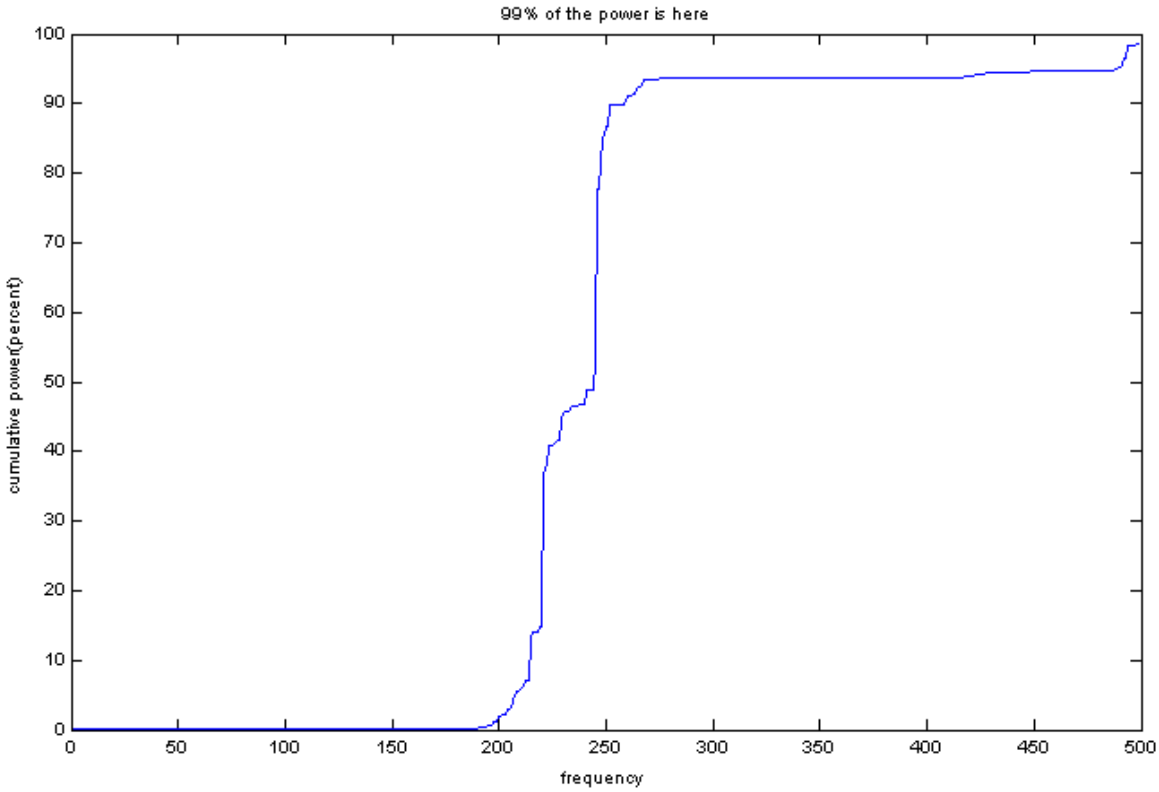
Instead of dealing with the funny numbers, we'll group together all the frequencies between zero and n, then run n from one to 22050:

```
% first, intitalize the loop:
n=[];
for j=1:22050
%find the frequencies less than j
%and look at P for only those frequencies
a=P(find(f<j));
%norm computes the length of the vector;
%norm(P) is the total power, and the 100 gives percent
n = [n 100*norm(a)/norm(P)];
end
%now look at the frequencies that make up 99 percent of the power
>> ind = find(n<99);
%and plot only for those frequencies
>> plot(n(ind))
```

For an example of how this works,

```
%find when the cumulative power is over 99 percent of total
ind = find(n>99);
%now check the first frequency for which this happens:
>> ind(1)
ans = 202
```

Ninety-nine percent of the power ios concentrated in frequencies from zero to 202Hz. Here's the graph of cumulative power:



The majority of power is concentrated between 190 and 250Hz, for this piece.

Lab Project: We have three different songs to experiment on: Sezen Aksu's sezen_44.wav, sampled at 44100 Hz., which is voice alone; K.D. Laing's 'Don't Let The Stars Get In Your Eyes' kd_44.wav which has voice and a bit of accompaniment, and, Sheila Chandra's Waiting', wait_44.wav, which also is voice and single instrument accompaniment. Finally, there's daily-life speech by actress Lucy Lawless, daily_11.wav, sampled at 11025Hz.

Pick several of these and analyze the cumulative power distribution. Keep in mind the graph below, which plots frequency on the horizontal scale, and, on the vertical scale, what sound pressure is required so that the ear can just hear the sound (threshhold of hearing). The lower the graph, the easier it is for the ear to hear at that frequency.

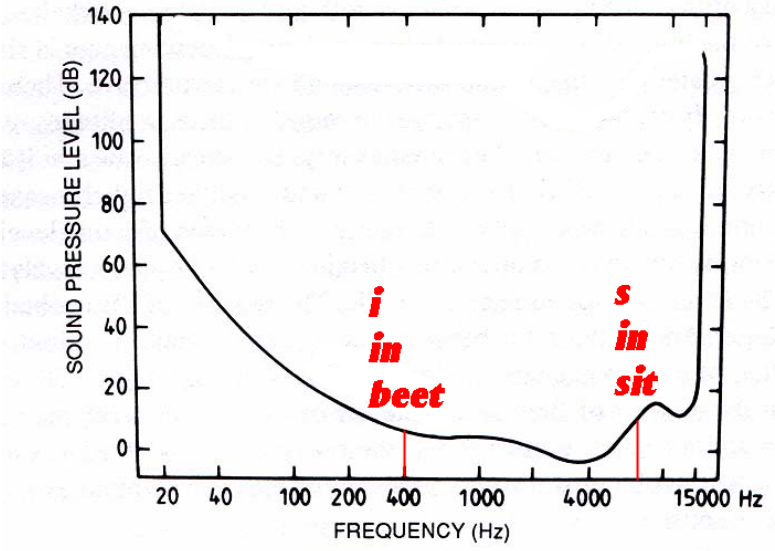
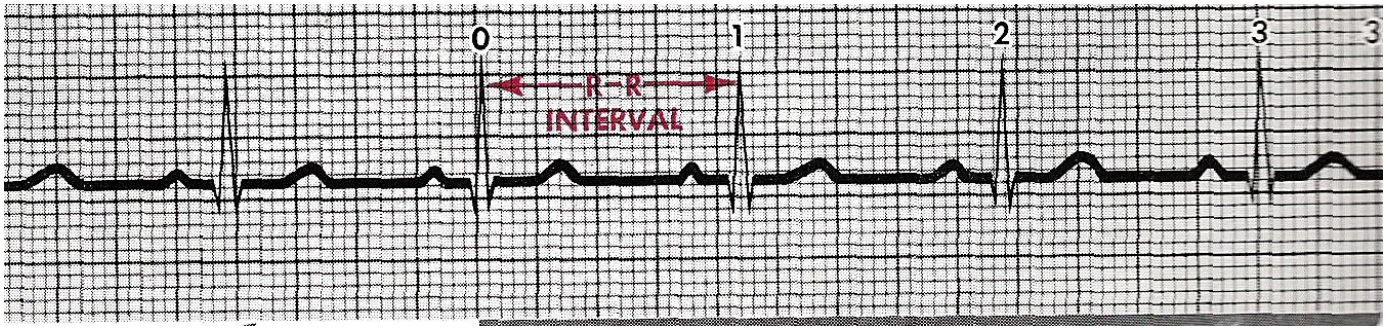


Figure 3.8 Audible frequency and sound level range.

Lab Project Use the information in the cumulative power distributions you did above, to compress the sounds. How does it sound, and, what factor of compression did you get?

Other researchers use power spectra in lots of ways; do a web search on the term "Power Spectrum" and see what you get. The power spectrum of energy in the sky gives astronomers a sense of how the early universe was formed. It's also used in image analysis. Check out the links!

In our biomedical research, the power spectrum was used very early in the papers of Akselrod et. al., Science 1981 Jul 10;213(4504):220-2, to analyze the heartbeat. They started with the so called RR-interval, the time between successive beats.



Above, electrocardiogram of a heart-healthy, resting human. If a person starts to exercise, the heartbeat speeds up, and the RR interval therefore gets smaller.

The point, though, is that body regulates the heartbeat in many other ways. If you're lying down and you stand up quickly, the blood all rushes down from the head and you faint and die -- or you wouldm, unless there were a mechanism to speed up the heartbeat. If you're exercising, you need to speed up the beat. If you're sleeping, the heart can slow down. The system that regulates the heart is called the autonomic nervous system. It must have a component that speeds up the heart -- the adrenergic system, after adrenalin, the chemical that the system uses. It is also called the sympathetic system. It must also have a system to slow things down -- the cholinergic system, after acetycholine; the system is also called the vagal system, after the nerve that performs the regulation. And, just to make things complete, it's also called parasympathetic, and, in some literature, muscarinic. Ah, medicine.

The actual regulation is quite complex, as it must be to keep us alive through all kinds of changes; it is still not completely understood. However, we can try to measure the various components, and how well tuned they are. The heart literature speaks of sympathovagal balance, to refer to this tuning.

And now we come to the main point: you can't check the health of an individual by sticking needles into people to check how their chemical and nerves are. Akselrod and her group showed that the sympathetic and the vagal systems used different parts of the power spectrum of the FFT of RR interval recordings. Many researchers (including us) are now trying to use that as means of getting real-time diagnostics.

Power Spectrum Analysis of Heart Rate Fluctuation: A Quantitative I of Beat-To-Beat Cardiovascular Control

Solange Akselrod; David Gordon; F. Andrew Ubel; Daniel C. Shar Barger; Richard J. Cohen

Science, New Series, Vol. 213, No. 4504 (Jul. 10, 1981), 220-222.

You can examine this issue with the heartbeat file BPM_042_2Hz.txt. The second column is time between beats; it is this you'll take the power from.

Entropy

Lab Project Total energy isn't the only measure of 'how much' signal there is. A quantity called *entropy* is a measure of how much information a signal contains.

You can get a feel for entropy by computing entropies, or plotting cumulative entropy distribution vs time, for different kinds of signals. For example, want.wav, Ricky Martin's Worl Cup song, is very repitituous, as is the constructed song mind.wav. Compare their entropy distributions to some of the free-form music above.

You can do the same for the Fourier Transform side. Instead of a cumulative power distribution graph, do a cumulative entropy graph of the fft's. What do they look like? And, how do they sound when compressing?

Phase

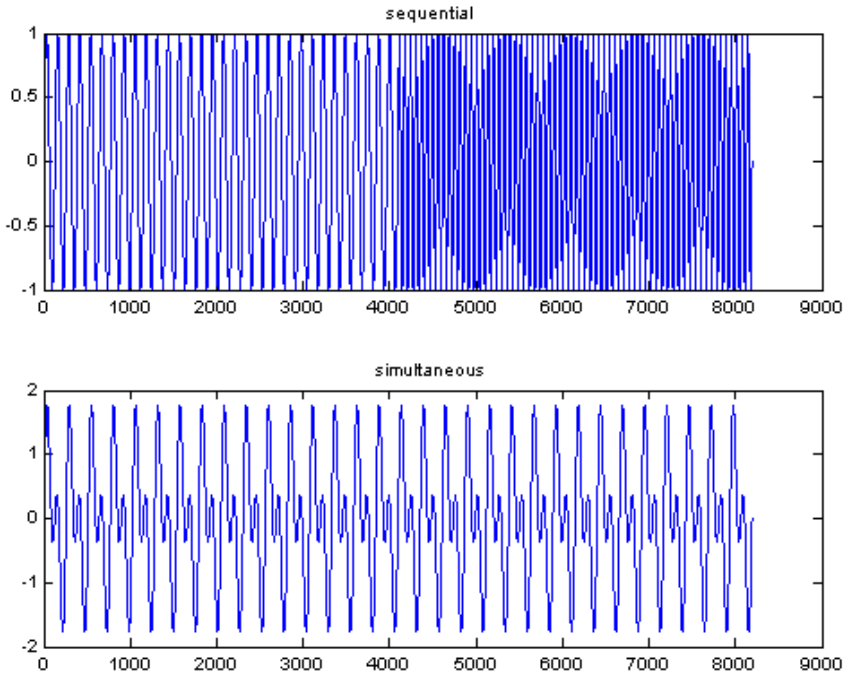
We've been acting as though the power spectrum of a piece of music contained everything in that music that we care about. It isn't, and here's an easy way to see it.

We went to Matlab, and created two sounds:

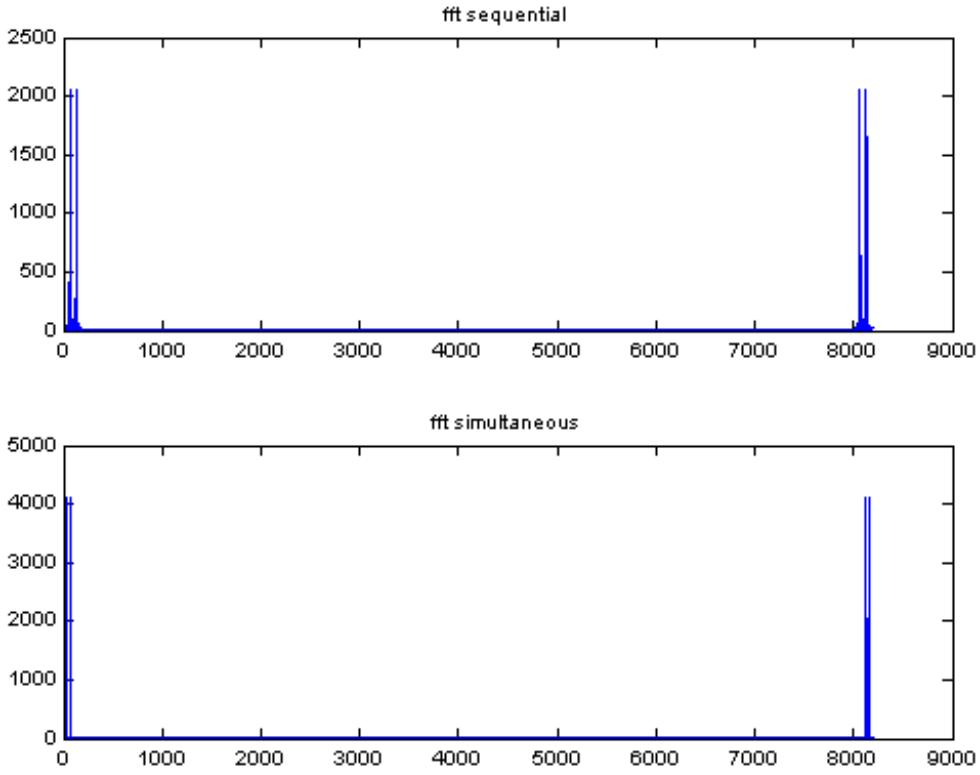
```
>> x=linspace(0,1, 4096);
>> y64=sin(2*pi*64*x);
>> y32=sin(2*pi*32*x);
>> y=[y32 y64];
>> t=linspace(0,1, 8192);
>> z=sin(2*pi*64*x)+ sin(2*pi*32*x);
```

And plot:

```
>> subplot(2, 1, 1), plot(y), title('sequential')
>> subplot(2, 1, 2), plot(z), title('simultaneous')
```



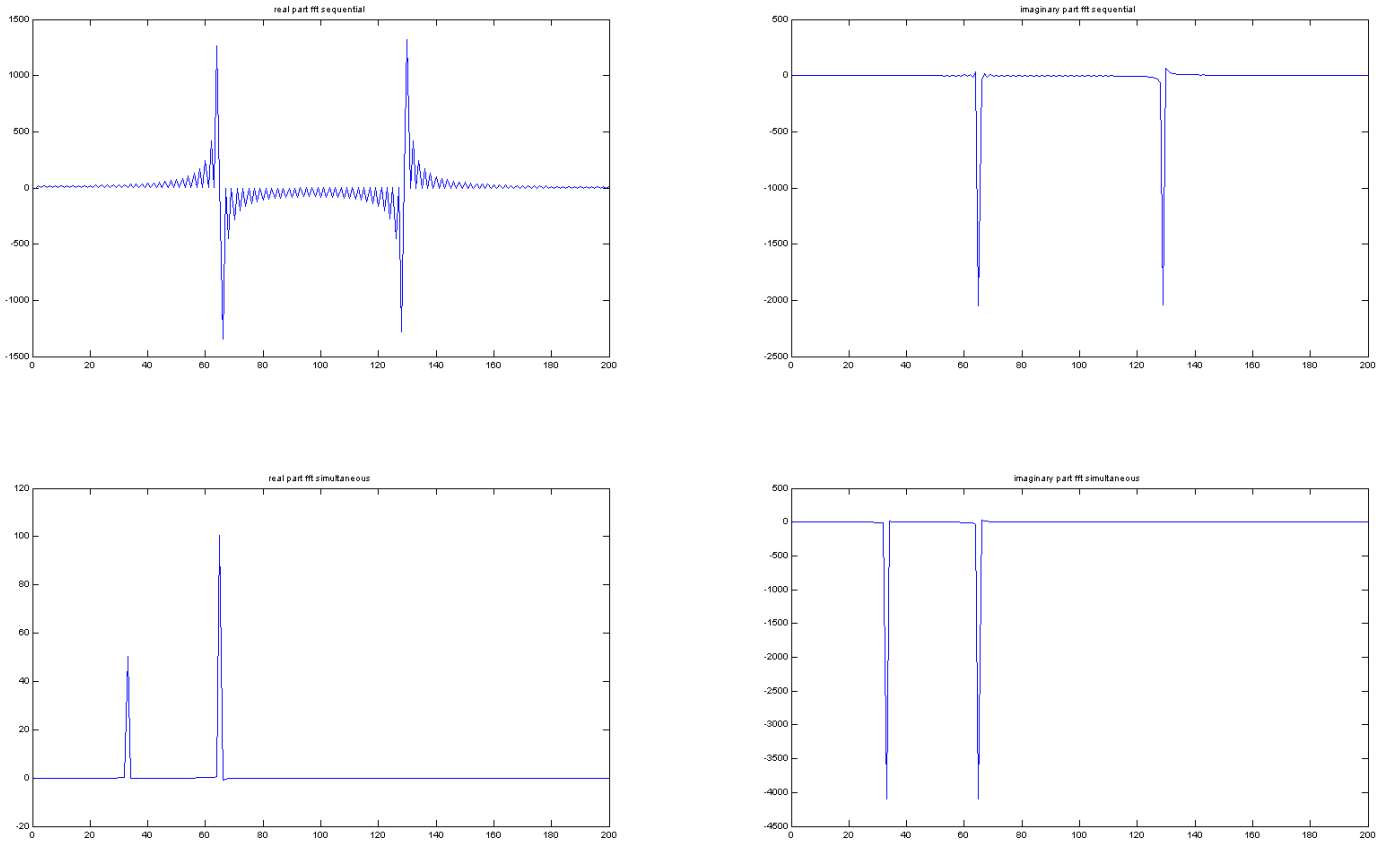
Thus, sequential has one sine after the other; simultaneous has them together. Here's the FFT's:



Not seeing a big lot of difference, there: the power spectrum can't detect the timing of events.

But the timing information can't be lost; after all, the IFFT regenerates the whole signal. Where'd it go? Into the real and imaginary parts of the FFT:

```
>> fy=fft(y);
>> fz=fft(z);
>>subplot(2, 2, 1), plot(real(fy(1:200))), title('real part fft sequential')
>> subplot(2, 2, 2), plot(imag(fy(1:200))), title('imaginary part fft sequential')
>> subplot(2, 2, 3), plot(real(fz(1:200))), title('real part fft simultaneous')
>> subplot(2, 2, 4), plot(imag(fz(1:200))), title('imaginary part fft simultaneous')
```



What's the deal? A complex number $z = x + iy$ can be written in polar form,

$$z = re^{i\theta} = r\cos\theta + ir\sin\theta$$

By the way – historical note: the term $r\sin\theta$ is called the *quadrature component* of the signal.

quad•ra•ture |ˈkwädrəˌ CH oŏr; -ˌ CH oŏr|
noun
1 Mathematics the process of constructing a square with an area equal to that of a circle, or of another figure bounded by a curve.
2 Astronomy the position of the moon or a planet when it is 90° from the sun as viewed from the earth.
3 Electronics a phase difference of 90° between two waves of the same frequency,

When you take the absolute value,

$$\sqrt{z\bar{z}} = \sqrt{re^{i\theta}re^{-i\theta}} = \sqrt{r^2} = r$$

The phase $e^{i\theta}$ is lost. And what does phase have to do with time? Watch this:

Let's start with a signal $f(n)$ defined on $\mathbf{Z}$. Shift it backwards three units in time: $g(n) = f(n+3)$. Now let's compute us a Fourier transform:

$$\hat{g}(\theta) = \sum_{n=-\infty}^{\infty} g(n)e^{-2\pi i\ n\theta} = \sum_{n=-\infty}^{\infty} f(n+3)e^{-2\pi i\ n\theta}$$

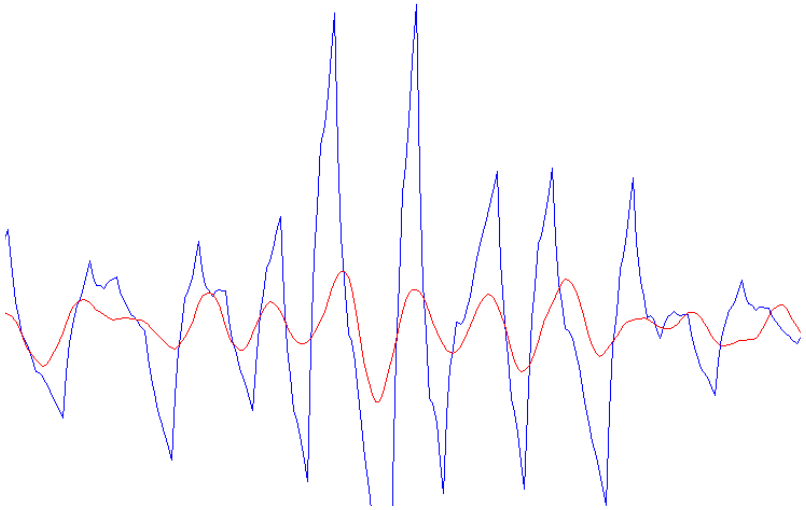
$$= \sum_{m=-\infty}^{\infty} f(m)e^{-2\pi i\ (m-3)\theta} = e^{-2\pi i\ 3\theta} \sum_{m=-\infty}^{\infty} f(m)e^{-2\pi i\ m\theta}$$

So that $\hat{g}(\theta) = e^{-2\pi i\ 3\theta}\hat{f}(\theta)$. The time-shift on f has been changed into a phase-shift in $\hat{f}$.

This may not seem like a big deal, but if you're doing digital music, it can cause big problems. If your mp3 compressor isn't programmed exactly right, it can cause phase shifts in parts of the music. When you play the music back, you hear pre-echoes: pieces of the music starting at the wrong time.

It's also a big deal for scientists working with heartbeats and RR-intervals. A phase shift can change the RR-interval, making the heart appear slow or fast.

Here's an example from our work. We start with a collection of RR-intervals (beats per minute, also known as BPM files) and we want to understand the physiologic mechanisms that underlie changes in the beat. We know that the two basic heart control mechanisms operate at different frequencies, so we want to isolate out the different frequency components in the BPM files. We're using a wavelet filter to do this (technically, a wavelet packet decomposition, with a Daubechies 2 wavelet, level four) and there's a problem: the wavelet changes phase. Worse yet, different frequencies are treated with different phases. This causes high frequencies and low frequencies to be shifted by different amounts:



Look at the first red peak; it trails the blue peak slightly. The peak is broad, so this is a LF wave, and the phase shift is negative. as you move right, the blue peaks narrow, and we're seeing HF events. The red peaks start matching or leading the blue peaks -- for HF, the phase shift is zero or positive. If this were music ----

Lab Project : For us, it suggests speculation and experiments. Load the Indigo Girls song night.wav, take the fft, then change the phases. One way is to multiply the first part of the fft by $e^{-\pi}$, and the second half by $e^{-\frac{\pi}{2}}$.

Here it is in Matlab:

```
ph=[(-1).*ones(1, 337400) (i).*ones(1, 337400)];
g=ph'.*f;
>> t=ifft(g);
>> wavwrite(t, 'phase.wav')
```

How's phase.wav sound? Can you do phase shift that aren't so awful? Say, cause a section of the music to pre-echo?

If you tried that at all, there must have come a time when you just wanted to grab a short segment of sound, and shift that around. There's a technique allowing you to do just that; it's called the Short Time Fourier Transform, or STFT. Here's a bit of background.

And there's a nice application. If you've ever messed with LP's or cassette tapes, you know that you can slow the sound by retarding the progress of the tape or LP. Then everything sounds deeper. But what if you wanted to slow the sound without deepening the sound? For example, I love Shakira's Estoy Aqui, but my Spanish isn't good enough to catch all the words. What if I could slow her down, so I could hear better?

How'd I do that? Well, with canned software, for one. For two, using STFT and phase adjustments ...

Lecture 2

FOURIER TRANSFORMS

Sampling

What if we lived in the world of The Matrix?.

Daniel Dennet, in his book on Consciousness, imagined just that, but he went a bit further. He asked, imagine if we were in the Matrix, and a computer was pumping data into us, to make is think we were free, living in the real world. How much data, how fast would it have to get into us, to sustain the illusion?

It's a fun computation, and after you do it, you see the supercomputers of The Matrix are just bad dreams: no real computer could get that much data, that fast.

But you also get a feel for what the pioneers of motion pictures had to do. After all, they were trying to create a (two-dimensional) version of reality. And the only tool they had was the picture. More accurately, a sequence of pictures, strung out on a long piece of film.

Talk about data rates! How many pictures per second would you need, to make people believe they were seeing real life?

We ran a little experiement, a clip of Marlene Dietrich's famous chanteuse scene with Gary Cooper, and made our own little films. They're titled md2.mov, md4.mov, md8.mov. Or you can view them from the CD, in the folder ch2/sampling. The clip md2.mov has two pictures per second, etc

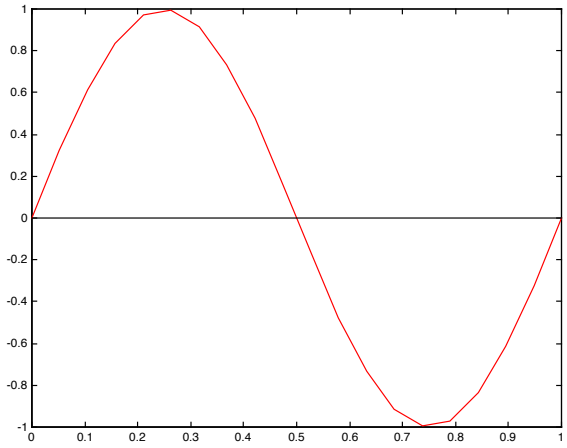
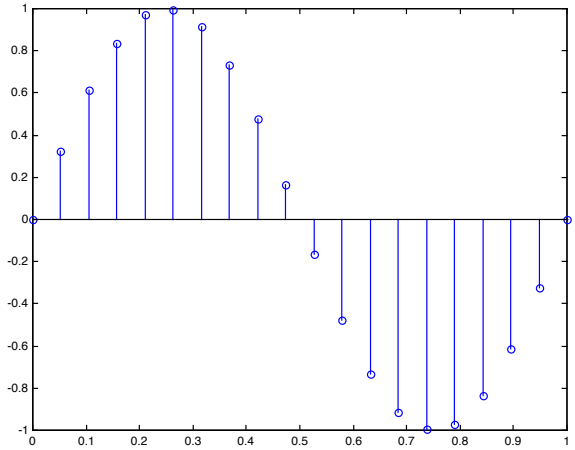
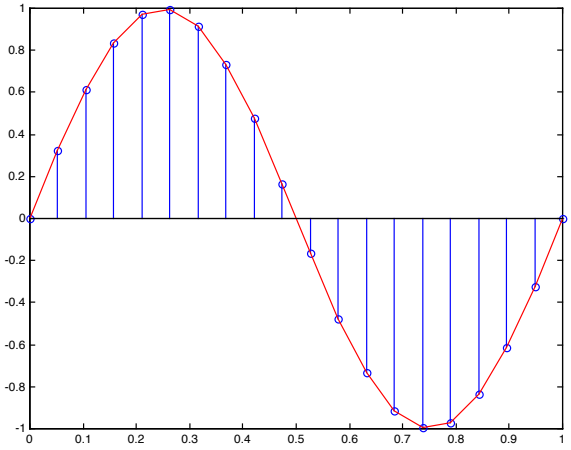
Play the clips. Of course md2 isn't going to fool anyone, but by md8, at 8 pictures per second, things seem pretty real.

I chose the Dietrich clip because there's a lot going on in the background behind Dietrich. People are fanning themselves, people are applauding. These are periodic motions. Try tracking some of the fans and some of the applause, in md2 through md8. Notice anything special?

In doing digital music, we have the same problems as the computers of the Matrix. Say you want to input one period of a sine wave, into a computer -- sin(2*pi*x) for x between zero and one. Well, it can't be done: there's an infinite number of points between zero and one. Take a computer with terabytes and terabytes of storage . . it'll never even get close to all that's stored in the numbers between zero and one.

(By the way, if you think the answer is a compression scheme, there's a nice theorem: most numbers between zero and one cannot be compressed).

The solution, the way to get music onto a computer, is the same solution the Hollywood producers used. Take snapshots of the function. This process is called sampling. And it isn't hard to see that if you sample at enough points, just like in the movies, things look real.

On the left, a graph of y(x) = sin(2*pi*x) for one period of the sine; on the right, the sin taken at only 20 points of the interval. Below, the two graphed together. Technically, what you have is y[j] = sin(2*pi*j/20) for j = 0, 1, 2, ... 19. Notice that the space between samples is (j+1)/20-j/20 = 1/20. The 20 represents the number of samples per second (assuming the x-axis here is time, and 0<x<1 represents one second!!). The number 20 is called the sampling frequency, and is usually denoted by an F or F with a subscript s. . It has units "samples per second". If F measures samples per second, 1/F measures the number of seconds which elapse between samples. Thus 1/F has the dimensions of a time, and it is usually written as T. It represents the time between samples. The "sampling times" themselves are jT = j/F. These are "equally-spaced" samples; we talk about non-equally spaced sampling at the end of the section.

In the function sin(2*pi*f*x), the variable f is called the frequency and is measured in "cycles per second" because, when x goes from zero to 1, 2*pi*f makes the sine go through f cycles. The number 2*pi*f is measured in "radians per second". When you do sampled functions, sin(2*pi*f*jT), f is now measured in "cycles per sample" and 2*pi*f*jT is radians per sample. Again, if T=1/20, then as j goes from 1 to 2, say, 2*pi*jT goes through 1/20 of a cycle of sine, and 2*pi*f*jT goes through f/20 cycles of a sine in one sample; hence cycles per sample.

Sampling is a form of compression. Instead of the pairs (x, sin(2*pi*x)) for an infinite number of x's, we have pairs (jT, sin(2*pi*jT)) for different j's. In fact, if we actually know the sampling frequency T, we can omit the jT altogether. This is the approach used in modern digital music; the sampling frequency is 44100 samples per second (an interesting number), or for professional equipment, 48,000 or 96,000 samples per second (we'll say more about these numbers later). This is the way music is encoded on a modern CD.

When the CD first came out, critics said, "you'll be able to hear the gaps between the music". It's a perfectly reasonable fear; something will be going on while we're not looking.

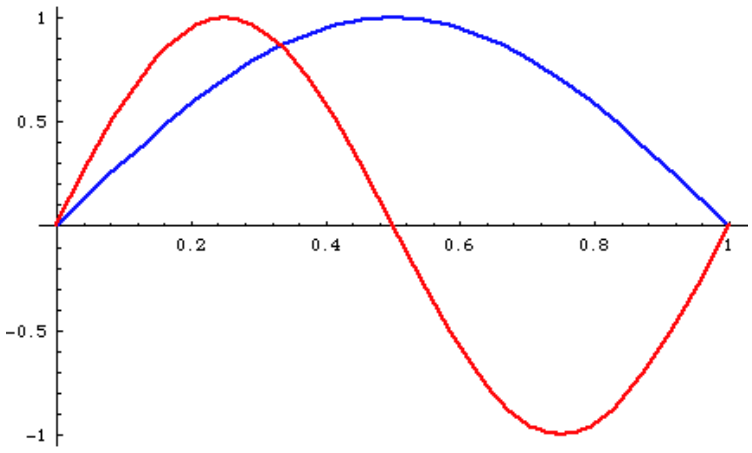
On the other hand, we sample all the time. When you're crossing a street, you look both ways. But you can't look both ways, so you look one way, then the other, then back again to see if anything has changed. You're sampling. While you're looking the other way, won't a car sneak up on you?

Well, no: not if the cars are travelling 35mph. We pretty much can see the street is clear and it's going to stay clear for awhile. On the other hand, if cars travelled 500 miles an hour, we couldn't take my eyes away. We might indeed miss something, while I was turned the other way.

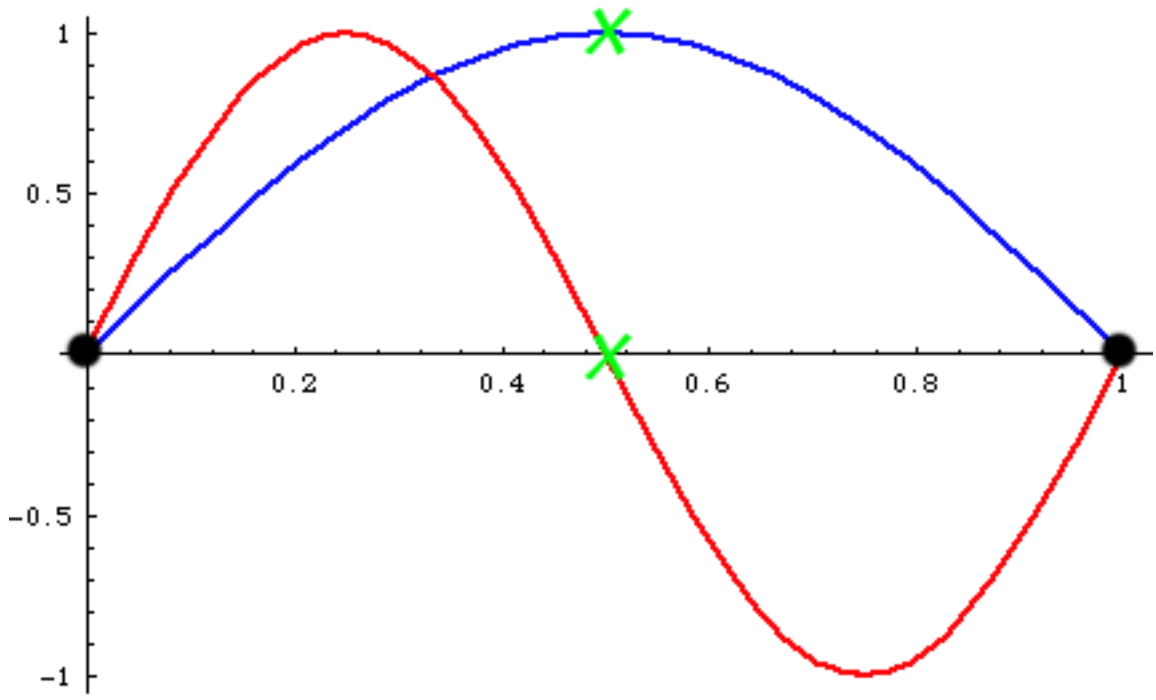
How fast you have to sample depends on how fast new events are coming in. The simplest way to model this for music is to look at sampling a sine wave, y = sin(2*pi*f*x). How fast must I sample it, in order to tell what the frequency is? (how small is T?).

My intuition is, the higher the frequency, the faster the sine is changing(the derivative is the speed of change, and y'=2*pi*f*cos(2*pi*f*x), which is proportional to the frequency f. So: to detect a frequency f, I figure I have to sample f times per second.

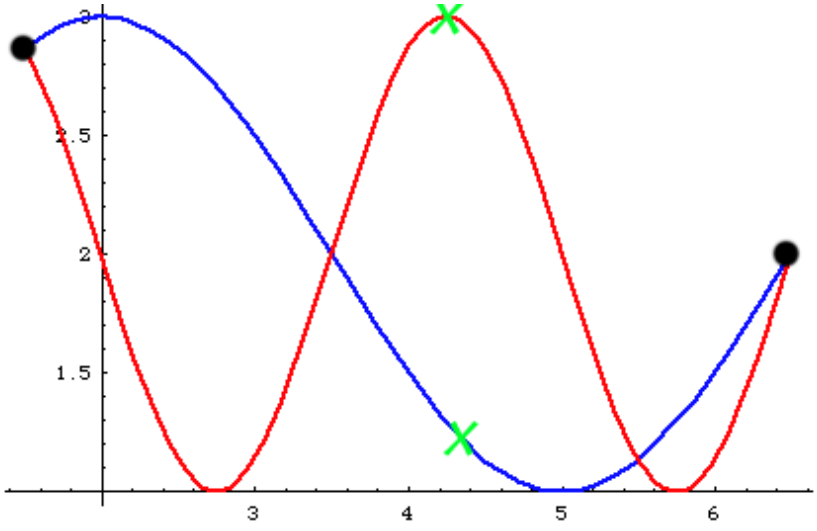
Which turns out to be wrong. Check out the picture below, in which y = sin(pi*x) is graphed in blue, and z = sin(2*pi*x) in red.



To be able to tell the difference, we need to sample . . well, once? No, I don't think so. If F=1, y[0/1] = z[0/1] so we'll need to sample another time to distinguish the two. Try T=1/2, F = 2:



There! If T = 1/2, j=0, 1 and we sample at 0/2, 1/2. Again y[0/2] = z[0/2], but y[1/2] ≠ z[1/2]. And, the picture below shows this isn't just for sines defined on 0<x<1.



So there's your sampling theorem: in order to capture the correct frequency of y=sin(2*pi*f*x), you have to choose a sampling rate of F greater than or equal to 2*f. 2*f is called the *Nyquist sampling frequency*, after the British mathematician Whittaker, who discovered it. Really, after the communications engineer Nyquist who rediscovered it. 1/T ≤ 2*f or T*f ≥ .5

If you fail to choose Nyquist, you'll miss out on some frequencies. But that turns out to be a deceptive way of speaking. Like when I'm crossing the street and I "miss seeing a car coming" is a deceptive way of saying "I'm dead."

We want to look at a couple of examples, to see what happens when we "miss" frequencies. The first goes back to the Dietrich movie clip. Compare md8 with md2 again, focusing on the gentleman who offers Dietrich a glass of champagne. First look at md8. The scene, right after Cooper salutes her and she tips her hat to him. The gentleman is fanning himself rather quickly. If you go to md2, however, the fanning almost disappears; it looks like a kind of fluttering, like the gentleman can't keep his hand steady.

The fan, however has not disappeared. One way of thinking about it is that the sampling has distorted the frequency of the fan motion. A high frequency fan motion has been made to appear like a lower frequency fan. Just like with our sines, y = sin(pi*x) appearing the same as z = sin(2*pi*x), when we sampled at T=1 instead of T=1/2.

This distortion is called *aliasing*; a high frequency looks like a lower one. You can predict it exactly. Here's the math:

Let's say you're sampling at a rate 2F; the highest frequency you can distinguish then is F. So what happens to a frequency F+f, when you sample at points j/2F? What does F+f alias into?

$$\sin\left(2\pi(F+f)\frac{j}{2F}\right) =$$
$$\sin\left(2\pi(F)\frac{j}{2F}\right)\cos\left(2\pi(f)\frac{j}{2F}\right) + \cos\left(2\pi(F)\frac{j}{2F}\right)\sin\left(2\pi(f)\frac{j}{2F}\right) =$$
$$\sin(j\pi)\cos\left(2\pi(f)\frac{j}{2F}\right) + \cos(j\pi(F))\sin\left(2\pi(f)\frac{j}{2F}\right) =$$
$$0\cdot\cos\left(2\pi(f)\frac{j}{2F}\right) + (-1)^j\sin\left(2\pi(f)\frac{j}{2F}\right) =$$
$$(-1)^j\sin\left(2\pi(f)\frac{j}{2F}\right)$$

Whereas if we sample a sine with frequency $F-f$ at the same j, you'll get the $(-1)^j\sin\left(2\pi(f)\frac{j}{2F}\right)$, but with an extra factor of -1. The values aren't the same, but the sound they make will be, and they'll appear to be the same frequency.

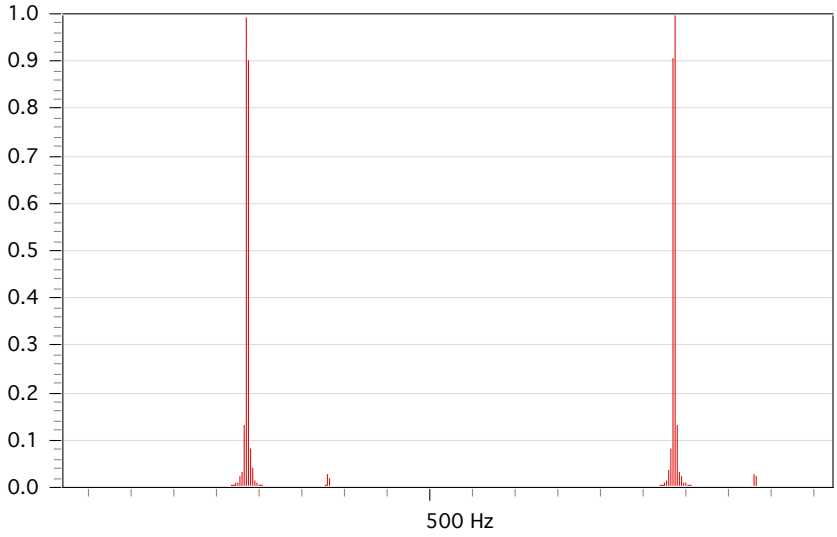
Which answers our question: a sine of frequency F+f aliases down to F-f.

For Dietrich, the aliasing left a gentleman without his fan. What's the effect on some music? To do this example, we construct a sound in Matlab.

```
x=linspace(0,1, 1024);
s1=sin(2*pi*457*x)+sin(2*pi*557*x);
y=linspace(0,1, 2048);
»s2=sin(2*pi*457*y)+sin(2*pi*557*y);
```

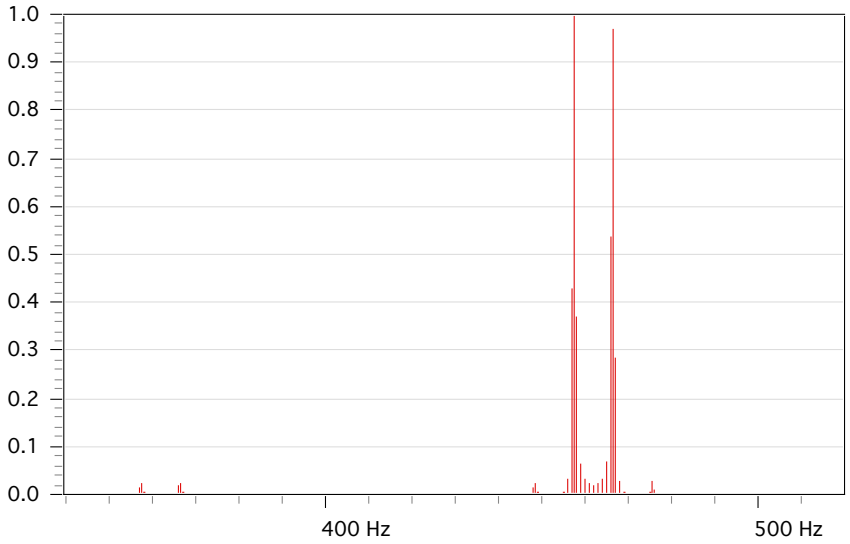
The sound s1 is a sum of sines, with frequencies 457hz, 557hz, sampled at 2F=1024. s2 is the same sound, except sampled at 2F=2048.

Now, when 2F=2048, F=1024 and the frequencies 457hz, 557hz are less than F, hence sampling doesn't alias them. Play 2048.wav, in the .ch2/sampling folder. And, while we're at it, here's the spectrum.



We see the two different frequencies, about equally spaced either side of 512hz.

Now try s1. Here, 2F=1024, so F=512, which means that 557hz = F+f where f=45hz. When you sample, this ought to be aliased down to F-f = 467hz. Here's the spectrum for s1:

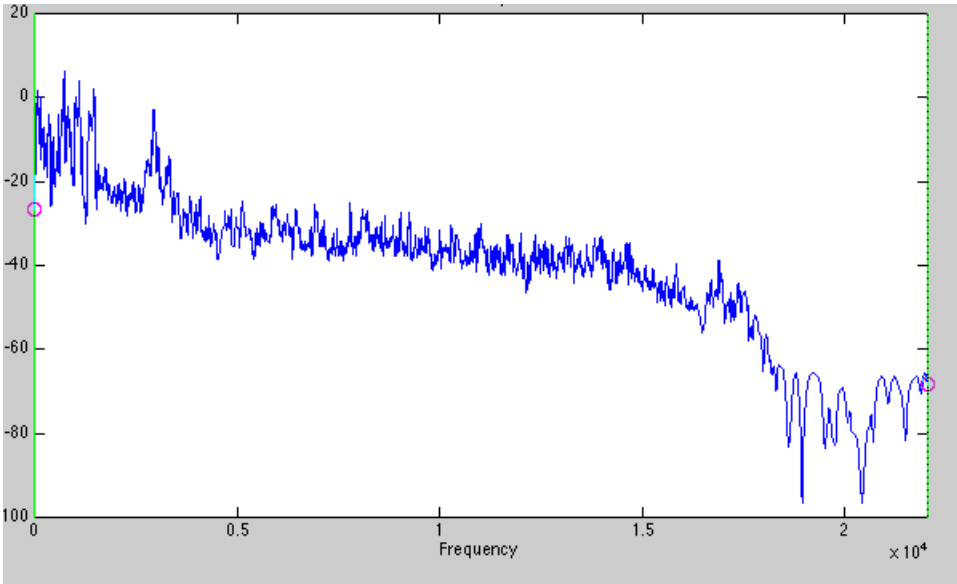


As predicted, but there's a little more to it than that: the the old frequency 447hz was well below the aliasing cut off, so it's unchanged. The 557hz sound used to be quite far away from 447, but now it's been aliased to 457hz, the two are within each other's *critical band*. Go ahead and play 1024.wav. There are now distinct beats. In a piece of music, this would be a disaster.

The moral here is that aliased frequencies don't disappear; they are changed into new frequencies that may never have been in the original sound. And when they get aliased, they may cause all sorts of problems, beats being only one.

Lab Problem: You can hear the effect with a series of clips constructed by starting with with the song, “Malaika”, by the Mahotella Queens, m32.wav or on the CD in /ch2/sampling. It is originally sampled at CD quality, 44100hz. we downsampled to a 32000hz mono signal, to make your work easier. Downsample to 16000hz, 4000hz, 2000hz and 1000hz. and listen to each. You'll especially enjoy the last.

Now look at what we hear. Find the spectrum of m32 and note there's little power past 2000hz, so we ought to be able to take a sampling rate of 4000hz, and keep the original sound. Play the sound downsampled to 4000hz and compare it with the oroginal. It sounds duller, as though it’s missing the high frequencies. And so it is; the spectrum picture was misleading. Here’s the original spectrum again, but this time plotted in decibels:



Ahem: oops! There are contributions to the spectrum all the way up to the maximum that 44100hz can give! And it seems the ear can hear some of this, too.

There's a moral in here. For example, say you were trying to compress sound. You look at the high frequency spectrum and say, “Look, the frequencies after about 4000hz fall off by 25db. We can cut those out, then go with a sampling rate of 8000hz. Let’s see, instead of 44100 samples per second, we’d have 8000. That’s like a compression down to 18% of the original! Wow.” Except it doesn’t sound so wow. Compression, alas, is not that simple.

Now try sampling at 4000hz and 2000hz. Note the spectrum of 4000hz has a nice peak at about 1100hz, and another at about 1500hz. Now compute the spectrum for the 2000hz downsampled music, where the highest frequency it can pick up is 1000hz. This means the frequencies at 1500hz and 1100hz that were in the previous have been aliased down to 900hz and 500hz. Check it out: sure enough, the spectrum shows strong peaks at around 900hz and 500hz. Compare with the original at 4000hz; see that there were no peaks at 900hz and 500hz, so these are now a result of aliasing.

Of course, the real test is the listening test: the music at 2000hz hasn’t just got an extra frequency or two; it sounds *weird* -- certainlly by comparison 8000hz, and even by comparison with 4000hz. The weirdness is due to something we haven’t quite encountered before:

the frequencies that get aliased down weren't just random noise; they were music. When they alias down they form form another kind of music. Strange music, because F+f becomes F-f so all the frequencies are reversed.

But music nonetheless. It's as though a second song were being played ontop of the first song.

The technical phrase is that the noise produced by aliasing is *correlated* with the original music. The ear picks up on that much more readily than it would just random sounds. There's a kind of hierarchy of badness here; some noise is worse than others.

The final moral here is that we can't afford aliasing, if we expect to do serious work with signals. Any signal has to be sampled at twice the highest frequency available, or the result will be corrupted. Since the ear can hear sounds up to about 20000hz, you need to sample at about 40000hz. The CD samples at 44100hz. Co-incidence?

The 44100hz standard for CD's came about because , after you sampling music, it had to be stored somewhere. The commercial system in the eighties that had the best storage capacity was professional videotape used, for example, to record TV shows. With a little futzing, one could convert the equipment to handle 44100 samples per second of sound. So the standard for CD players was born.

But there's another issue. Any recording session involves microphones, and those microphones pick up noise in the air, which often has frequencies much higher than 20000hz. If you don't eliminate the noise somehow, it will alias back down into all sorts of places you don't really want it. For this reason, professional recording begins with filtering: you pass the music though a filter, which eliminates all frequencies above 20000hz. At this point, the sound is sampled, and there's no aliasing.

In practice, it gets very complicated. First of all, the filter has to be applied before the sound is sampled. That means the signal isn't digital yet, so the filter has to be analog. These are hard to implement, and in fact they can't simply pass all frequencies below 20000hz, unchanged, and cut off all frequencies above 20000hz, completely. There needs to be a little leeway. That's why 44100 is better than 40000; it gives leeway between 22050 and 20000.

The next problem is with the sampling itself. We described it very mathematical -- f(jT) and it looks very clean. But in fact it has to be done, again, by analog circuits. Resistors and capacitors and diodes and transistors. And none of them are precise and mathematical. A capacitor takes time to discharge, and time to recharge.

One could try to optimize sampling by sampling *adaptively*. If a musical piece had only human voice for a while, then the instruments came in, the part with voice doesn't have high frequencies, so it could be sampled at a low rate, after which the high sampling rate for the instruments is used. You could compress music this way; probably it wouldn't even be very hard to code. See the article "sampling.pdf" in the c2/sampling folder of the CD for a modern application.

And, while we're at it . . . we've talked about the need to be careful when you sample music. But there's another field where no-one cares how people sound: the telephone! If you accept the idea that the telephone will be used to transmit voice, what's the proper frequency to sample?

Lecture 2

FOURIER TRANSFORMS

Reconstruction

Say we sample f at a rate $1/T$. If we expect to avoid aliasing, then f can't have frequencies higher than the Nyquist frequency, $F = 1/2T$. This means $\hat{f}(\xi) = 0$ if $|\xi| \geq 1/2T$. Such a function is called band-limited. The band can be described by the indicator function, $I(\xi) = \chi_{[-\frac{1}{2},\frac{1}{2}]}(\xi)$, which is zero for $|\xi| \geq 1/2$ and 1 otherwise. We define

$$\operatorname{sinc}(x) = \widehat{I}(x) = \int_{-\infty}^{\infty} I(\xi) e^{2\pi i x\xi}\, d\xi = \int_{-\frac{1}{2}}^{\frac{1}{2}} 1\; e^{2\pi i x\xi}\, d\xi$$

$$= \frac{1}{2\pi i x}\left[e^{\pi i x\xi} - e^{-\pi i x\xi}\right] = \frac{1}{\pi x}\left[\frac{e^{\pi i x\xi} - e^{-\pi i x\xi}}{2i}\right] = \frac{\sin(\pi x)}{\pi x}$$

Theorem Assume f is band-limited: $\hat{f}(\xi) = 0$ if $|\xi| \geq F$. Let $T = 1/2F$. Then (Whittaker Reconstruction Formula):

$$f(x) = \sum_{n=-\infty}^{\infty} f(nT)\, \operatorname{sinc}\left(\frac{x - nT}{T}\right)$$

Remark 1 This means in particular, if there is no aliasing, then f can be completely recovered from its samples – nothing is lost by sampling.

Remark 2 Notice that $\operatorname{sinc}(0)$ is undefined, but extending, $\lim_{x\to 0} \operatorname{sinc}(x) = 1$. Moreover, $\operatorname{sinc}(j) = 0$ if $j \neq 0$. Hence,

$$\sum_{n=-\infty}^{\infty} f(nT)\, \operatorname{sinc}\left(\frac{jT - nT}{T}\right) =$$

$$\sum_{n\neq j} f(nT)\, \operatorname{sinc}\left(\frac{jT - nT}{T}\right) + f(jT)\, \operatorname{sinc}\left(\frac{jT - jT}{T}\right) = f(jT)$$

Whatever else the Whittaker Theorem does, it gives the right answer on the points jT. Functions like $\operatorname{sinc}\left(\frac{x-nT}{T}\right)$ are called interpolation functions – they're 1 at the proper places and zero elsewhere.

Proof of the Whittaker Theorem: The function $\hat{f}(\xi)$ is zero outside the interval $[-F, F]$, so we can reconstruct $\hat{f}$ in terms of its Fourier series,

$$\hat{f}(\xi) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n\xi/2F}; \qquad c_n = \frac{1}{2F}\int_{-F}^{F} e^{-2\pi i n\xi/2F}\hat{f}(\xi)\, d\xi$$

However, as $\hat{f}$ is band-limited, the integral defining c_n can be extended, giving us

$$c_n = \frac{1}{2F}\int_{-F}^{F} e^{-2\pi i n\xi/2F}\hat{f}(\xi)\ d\xi = c_n = T\int_{-\infty}^{\infty} e^{-2\pi i n T\xi}\hat{f}(\xi)\ d\xi$$

But this is almost an inverse Fourier transform of $\hat{f}$. The only difference is, the exponential should have a $+\pi i n T\xi$ instead of a $-\pi i n T\xi$. This means that the inverse Fourier transform doesn't recover f; instead we get

$$c_n = Tf(-nT)$$

Whence,

$$\hat{f}(\xi) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n\xi/2F} = \sum_{n=-\infty}^{\infty} Tf(-nT)e^{2\pi i n T\xi}$$

$$= T\sum_{n=-\infty}^{\infty} f(nT)e^{-2\pi i n T\xi}$$

But,

$$f(x) = \int_{-\infty}^{\infty}\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi = \int_{-F}^{F}\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi = \int_{-\infty}^{\infty} I(\frac{\xi}{2F})\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi$$

$$= \int_{-\infty}^{\infty} I(\frac{\xi}{2F})\ T\sum_{n=-\infty}^{\infty} f(nT)e^{-2\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

$$= T\sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\frac{\xi}{2F})\ e^{-\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

If we change variables, to $\phi = \xi/2F$, we get

$$f(x) = T2F\sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\phi)\ e^{-2\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

$$= \sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\phi)\ e^{2\pi i\frac{\phi}{T}(x-nT)}\ d\xi$$

$$= \sum_{n=-\infty}^{\infty} f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$$

Remark 3 The Whittaker theorem is of use mostly for doing theory; in practical reconstruction of a signal, it isn't so good. Let's do a small computation; imagine that you have a three-minute piece of music on a CD. It's been sampled at 44100hz, , and now you want to play it back, at the same rate. Once every 1/44100 of a second, you have to compute $\sum f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$. Since the music is three minutes long, that's (3min)(60 sec/min)(44100 samples/sec) = 7,938,000 terms in the sum:

at least it isn't infinite, as it looked. Each involves a multiplication and a division, so by the time you're done, one second of music requires about a megabyte of storage and about 10^{12} floating point operations per second – what's commonly called a teraflop. Teraflop speeds are just being achieved now, in the latest generation of supercomputers.

This suggests that the Whittaker reconstruction theorem might not be really practical in cheap portable MP3 players.

Remark 4 We need a replacement for the Whittaker Theorem. The first step is to notice that there's nothing special about the function $I(\xi)$; any function would work as long as it is 1 where $\hat{f} \neq 0$ and zero when $\hat{f} = 0$.

In fact the $I(\xi)$ function has some serious disadvantages. As the sum can involve millions of terms, which have to stored in memory and then computed, we'd like to replace it with a smaller number of terms.

In practice, this means dropping the small terms. So, we hope that
$\sum f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$ has terms which get small real fast. Now $f(nT)$ won't be small; it's the amplitude of the music signal. Can we hope that $\mathrm{sinc}\left(\frac{x-nT}{T}\right)$ is small when n is large? Let's take an easy example; let's take $x = .1$. How big is $\mathrm{sinc}\left(\frac{.1-nT}{T}\right)$? If we ignore contributions from the sine term in the numerator, sinc is like $1/\pi x$, which makes my term approximately like $1/n$.

So how fast does $\sum 1/n$ get where it's going, compared to say $\sum 1/n^2$? Say you have to sum to about seven million terms, as in music example.

$$\sum_{n=1}^{7\cdot 10^6} \frac{1}{n} = 16.339; \qquad \sum_{n=1}^{7\cdot 10^6} \frac{1}{n^2} = 1.645$$

Now let's say we want to approximate these sums by their first hundred terms.

$$\sum_{n=1}^{100} \frac{1}{n} = 5.187; \qquad \sum_{n=1}^{100} \frac{1}{n^2} = 1.635$$

The relative error of a computation is measured as

$$\frac{|\ \text{true value} - \text{approximation}\ |}{|\ \text{true value}\ |}$$

For $\sum 1/n$ the relative error, then, is 68.25%, while for $\sum 1/n^2$ the relative error is .61%.

The moral here is that reconstruction via the sinc function is impractical; in practice, other reconstruction techniques are employed.

Lab Problem One approach is to simply force the series to converge faster by truncating it. Replace

$$\sum_{j=-\infty}^{\infty} \quad \text{by} \quad \sum_{j=-M}^{M}$$

Lecture 2

FOURIER TRANSFORMS

Reconstruction

Say we sample f at a rate $1/T$. If we expect to avoid aliasing, then f can't have frequencies higher than the Nyquist frequency, $F = 1/2T$. This means $\hat{f}(\xi) = 0$ if $|\xi| \geq 1/2T$. Such a function is called band-limited. The band can be described by the indicator function, $I(\xi) = \chi_{[-\frac{1}{2},\frac{1}{2}]}(\xi)$, which is zero for $|\xi| \geq 1/2$ and 1 otherwise. We define

$$\operatorname{sinc}(x) = \widehat{I}(x) = \int_{-\infty}^{\infty} I(\xi) e^{2\pi i x\xi}\, d\xi = \int_{-\frac{1}{2}}^{\frac{1}{2}} 1\; e^{2\pi i x\xi}\, d\xi$$

$$= \frac{1}{2\pi i x}\left[e^{\pi i x\xi} - e^{-\pi i x\xi}\right] = \frac{1}{\pi x}\left[\frac{e^{\pi i x\xi} - e^{-\pi i x\xi}}{2i}\right] = \frac{\sin(\pi x)}{\pi x}$$

Theorem Assume f is band-limited: $\hat{f}(\xi) = 0$ if $|\xi| \geq F$. Let $T = 1/2F$. Then (Whittaker Reconstruction Formula):

$$f(x) = \sum_{n=-\infty}^{\infty} f(nT)\, \operatorname{sinc}\left(\frac{x - nT}{T}\right)$$

Remark 1 This means in particular, if there is no aliasing, then f can be completely recovered from its samples – nothing is lost by sampling.

Remark 2 Notice that $\operatorname{sinc}(0)$ is undefined, but extending, $\lim_{x\to 0} \operatorname{sinc}(x) = 1$. Moreover, $\operatorname{sinc}(j) = 0$ if $j \neq 0$. Hence,

$$\sum_{n=-\infty}^{\infty} f(nT)\, \operatorname{sinc}\left(\frac{jT - nT}{T}\right) =$$

$$\sum_{n\neq j} f(nT)\, \operatorname{sinc}\left(\frac{jT - nT}{T}\right) + f(jT)\, \operatorname{sinc}\left(\frac{jT - jT}{T}\right) = f(jT)$$

Whatever else the Whittaker Theorem does, it gives the right answer on the points jT. Functions like $\operatorname{sinc}\left(\frac{x-nT}{T}\right)$ are called interpolation functions – they're 1 at the proper places and zero elsewhere.

Proof of the Whittaker Theorem: The function $\hat{f}(\xi)$ is zero outside the interval $[-F, F]$, so we can reconstruct $\hat{f}$ in terms of its Fourier series,

$$\hat{f}(\xi) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n\xi/2F}; \qquad c_n = \frac{1}{2F}\int_{-F}^{F} e^{-2\pi i n\xi/2F}\hat{f}(\xi)\, d\xi$$

However, as $\hat{f}$ is band-limited, the integral defining c_n can be extended, giving us

$$c_n = \frac{1}{2F}\int_{-F}^{F} e^{-2\pi i n\xi/2F}\hat{f}(\xi)\ d\xi = c_n = T\int_{-\infty}^{\infty} e^{-2\pi i n T\xi}\hat{f}(\xi)\ d\xi$$

But this is almost an inverse Fourier transform of $\hat{f}$. The only difference is, the exponential should have a $+\pi i n T\xi$ instead of a $-\pi i n T\xi$. This means that the inverse Fourier transform doesn't recover f; instead we get

$$c_n = Tf(-nT)$$

Whence,

$$\hat{f}(\xi) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n\xi/2F} = \sum_{n=-\infty}^{\infty} Tf(-nT)e^{2\pi i n T\xi}$$

$$= T\sum_{n=-\infty}^{\infty} f(nT)e^{-2\pi i n T\xi}$$

But,

$$f(x) = \int_{-\infty}^{\infty}\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi = \int_{-F}^{F}\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi = \int_{-\infty}^{\infty} I(\frac{\xi}{2F})\hat{f}(\xi)e^{2\pi i x\xi}\ d\xi$$

$$= \int_{-\infty}^{\infty} I(\frac{\xi}{2F})\ T\sum_{n=-\infty}^{\infty} f(nT)e^{-2\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

$$= T\sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\frac{\xi}{2F})\ e^{-\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

If we change variables, to $\phi = \xi/2F$, we get

$$f(x) = T2F\sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\phi)\ e^{-2\pi i n T\xi}e^{2\pi i x\xi}\ d\xi$$

$$= \sum_{n=-\infty}^{\infty} f(nT)\int_{-\infty}^{\infty} I(\phi)\ e^{2\pi i\frac{\phi}{T}(x-nT)}\ d\xi$$

$$= \sum_{n=-\infty}^{\infty} f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$$

Remark 3 The Whittaker theorem is of use mostly for doing theory; in practical reconstruction of a signal, it isn't so good. Let's do a small computation; imagine that you have a three-minute piece of music on a CD. It's been sampled at 44100hz, , and now you want to play it back, at the same rate. Once every 1/44100 of a second, you have to compute $\sum f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$. Since the music is three minutes long, that's (3min)(60 sec/min)(44100 samples/sec) = 7,938,000 terms in the sum:

at least it isn't infinite, as it looked. Each involves a multiplication and a division, so by the time you're done, one second of music requires about a megabyte of storage and about 10^{12} floating point operations per second – what's commonly called a teraflop. Teraflop speeds are just being achieved now, in the latest generation of supercomputers.

This suggests that the Whittaker reconstruction theorem might not be really practical in cheap portable MP3 players.

Remark 4 We need a replacement for the Whittaker Theorem. The first step is to notice that there's nothing special about the function $I(\xi)$; any function would work as long as it is 1 where $\hat{f} \neq 0$ and zero when $\hat{f} = 0$.

In fact the $I(\xi)$ function has some serious disadvantages. As the sum can involve millions of terms, which have to stored in memory and then computed, we'd like to replace it with a smaller number of terms.

In practice, this means dropping the small terms. So, we hope that
$\sum f(nT)\ \mathrm{sinc}\left(\frac{x-nT}{T}\right)$ has terms which get small real fast. Now $f(nT)$ won't be small; it's the amplitude of the music signal. Can we hope that $\mathrm{sinc}\left(\frac{x-nT}{T}\right)$ is small when n is large? Let's take an easy example; let's take $x = .1$. How big is $\mathrm{sinc}\left(\frac{.1-nT}{T}\right)$? If we ignore contributions from the sine term in the numerator, sinc is like $1/\pi x$, which makes my term approximately like $1/n$.

So how fast does $\sum 1/n$ get where it's going, compared to say $\sum 1/n^2$? Say you have to sum to about seven million terms, as in music example.

$$\sum_{n=1}^{7\cdot 10^6} \frac{1}{n} = 16.339; \qquad \sum_{n=1}^{7\cdot 10^6} \frac{1}{n^2} = 1.645$$

Now let's say we want to approximate these sums by their first hundred terms.

$$\sum_{n=1}^{100} \frac{1}{n} = 5.187; \qquad \sum_{n=1}^{100} \frac{1}{n^2} = 1.635$$

The relative error of a computation is measured as

$$\frac{|\ \text{true value} - \text{approximation}\ |}{|\ \text{true value}\ |}$$

For $\sum 1/n$ the relative error, then, is 68.25%, while for $\sum 1/n^2$ the relative error is .61%.

The moral here is that reconstruction via the sinc function is impractical; in practice, other reconstruction techniques are employed.

Lab Problem One approach is to simply force the series to converge faster by truncating it. Replace

$$\sum_{j=-\infty}^{\infty} \quad \text{by} \quad \sum_{j=-M}^{M}$$

where M is your favorite number. This has the advantage that it's easy to calculate the error – it's less than the error in approximating $\hat{f}$ by its truncated Fourier series. Take the piece of music sampled at 44100Hz, love.wav, 'A Love Before Time' from Crouching Tiger, Hidden Dragon, and reconstruct using the sinc function, but truncate the sum to some reasonable amount.

This is tricky to realize in practice, because Matlab won't deal with real numbers x; it only deals with samples. If you use the same sampling frequency as the original, will you get back the original sound? Try it.

The alternative is to compute the reconstructed f at extra points in between the original. Say that between each of the old points, you insert three, equally spaced, new points using the truncated reconstructor. You'd now have 176,400Hz. You can now play the sound. However, this isn't particularly helpful: the music on a computer will play by sending samples to a sound card. The sound card does its **own** reconstruction in order to play the sound.

Unless you are an engineer and can design your own audio equipment, it's hard to get around this. What you CAN do is compare spectra of the original and the reconstructed sounds. What do you see?

Another approach is to notice that sinc is the transform of $I(\xi)$; can we change I, then? Yes; all I had to do was satisfy

$$I(\xi) = \begin{cases} 1 & \text{if } f(\xi) \neq 0 \\ 0 & \text{if } f(\xi) = 0 \end{cases}$$

There are many functions which do this. For example, we could have a function $J(\xi)$ with

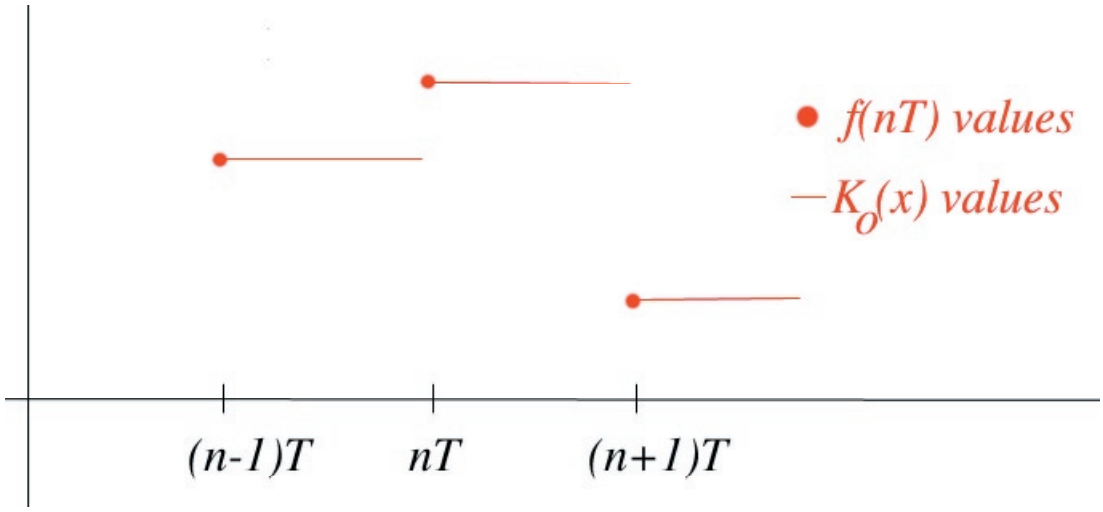
$$J(\xi) = \begin{cases} 1 & \text{if } \xi \in [-F,\ F] \\ 0 & \text{if } \xi \notin [-2F,\ 2F] \end{cases}$$

In between f and $2F$, J could be a straight line, or any nice function. Go ahead and check – in Matlab – that such a J will have a transform that goes to zero much faster than the sinc function.

The easiest reconstructor to design is the "zero-order hold" we looked at earlier. The terminology comes from the electrical engineers; they think of the $f(nT)$ as voltages that you have to do something with – multiply or multiply and add or whatever. The zero order hold takes the signal $f(nT)$ and just holds that value as the time goes from nT to $(n+1)T$. In terms like our other reconstructors, define $K_o(x)$ as

$$K_o(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{if } \xi \notin [0,\ 1) \end{cases}$$

Then our reconstructor is $\sum f(nT)\ K_o\left(\frac{x-nT}{T}\right)$. Here's a picture of what it does.



The problem is that this replaces f by a constant function. To go from one sampling interval to the next, the reconstructed f has to jump. These jumps produce problems with the frequencies in the reconstructed sound. For example, start with a function f with $f((n-1)T) = 0$; $f(nT) = 1$; $f((n+1)T) = 0$(such a function is called a *unit impulse function*; its transform the *'impulse response'*. Then the reconstructed f has the constant value 1 on the interval $[\ nT,\ (n+1)T\ ]$. When I take its Fourier transform, I get:

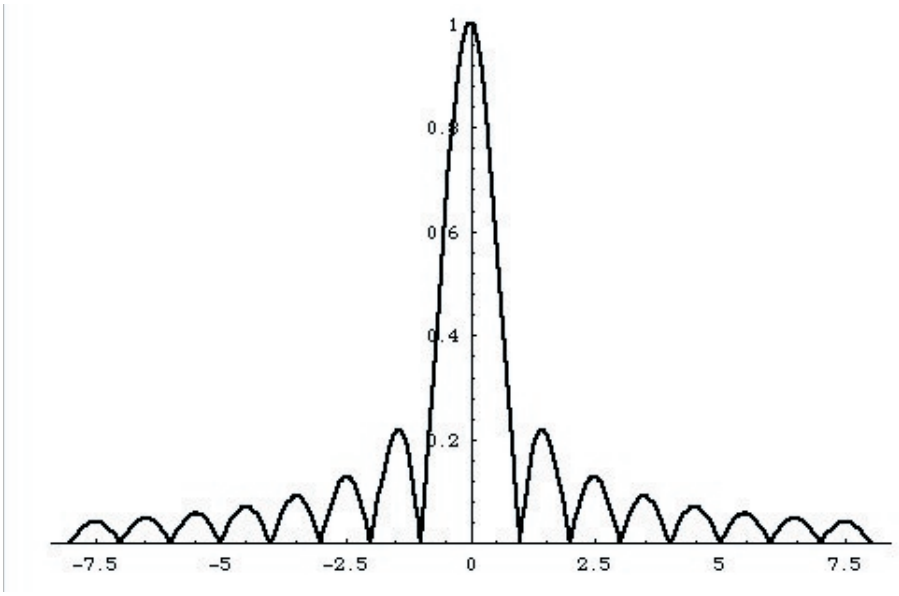
$$\int_{nT}^{(n+1)T} e^{-2\pi i x\xi}\mathbf{1}(x)\ dx = (x->y+nT)\ e^{-2\pi inT\xi}\int_0^T e^{-2\pi iy\xi}\mathbf{1}(y)\ dy$$

$$= (y->zT)\ Te^{-2\pi inT\xi}\int_0^1 e^{-2\pi izT\xi}\mathbf{1}(z)\ dz$$

$$= \frac{Te^{-2\pi inT\xi}}{-2\pi iT\xi}\left[e^{-2\pi iT\xi} - e^0\right] = \frac{e^{-2\pi inT\xi}}{-2\pi i\xi}\left[e^{-2\pi iT\xi} - e^0\right]\frac{e^{\pi iT\xi}}{e^{\pi iT\xi}}$$

$$= \frac{e^{-\pi i(n+\frac{1}{2})T\xi}}{\pi\xi}\left[\frac{e^{-\pi iT\xi} - e^{\pi iT\xi}}{-2i}\right] = \frac{\sin(\pi T\xi)}{\pi\xi}$$

The absolute value of this graphs as follows:



Note this function falls off as $\frac{1}{x}$. This means that reconstructing f with a zero order sample-and-hold adds high frequency noise, noise which decays very slowly. Again, demonstrate this in Matlab by computing the spectrum of a reconstructed f sampled at 176,400Hz, and then comparing that with the original spectrum.

For this reason, every real system that uses a zero-order sample and hold reconstructor follows it with a high frequency filter.

Finally, one note. High end audio systems use reconstructors that interpolate extra points, and give signals sampled at 176,400Hz. Here's an article from the Levinson/Madrigal website, or look in the folder on CD, ch2/reconstruction/oversample.pdf.

Lecture 2

FOURIER TRANSFORMS

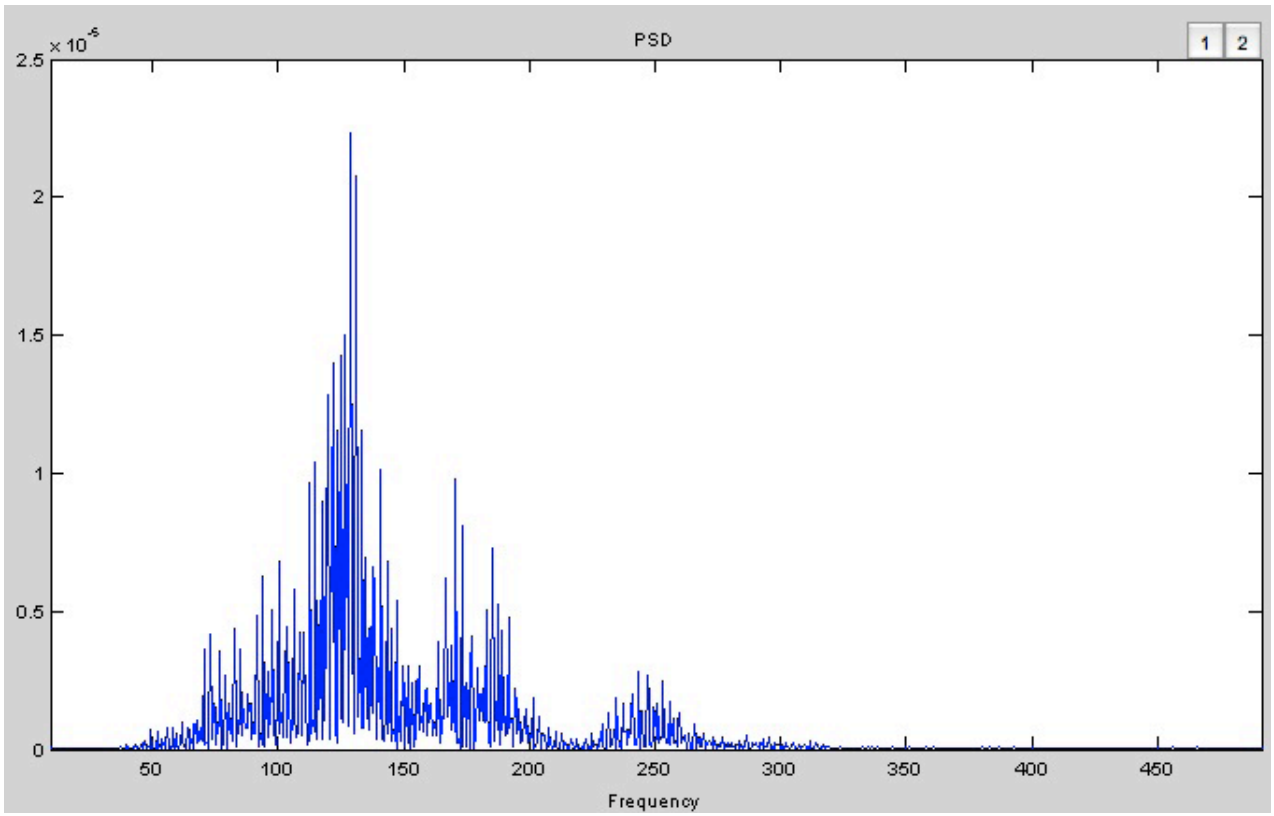
The Heart of the FFT

The FFT is a great tool for producing numbers but -- and you probably heard this before -- a major purpose of doing computations is to get insight into a problem, not numbers. So -- once you have a FFT, what does it mean? What understanding does it offer?

Let's take a very simple example: a heartbeat, as heard through a stethoscope, beat.wav. It's been sampled at 8012Hz. Import it into Matlab, take the FFT, and plot the absolute value. While you're at it, why not plot the signal?

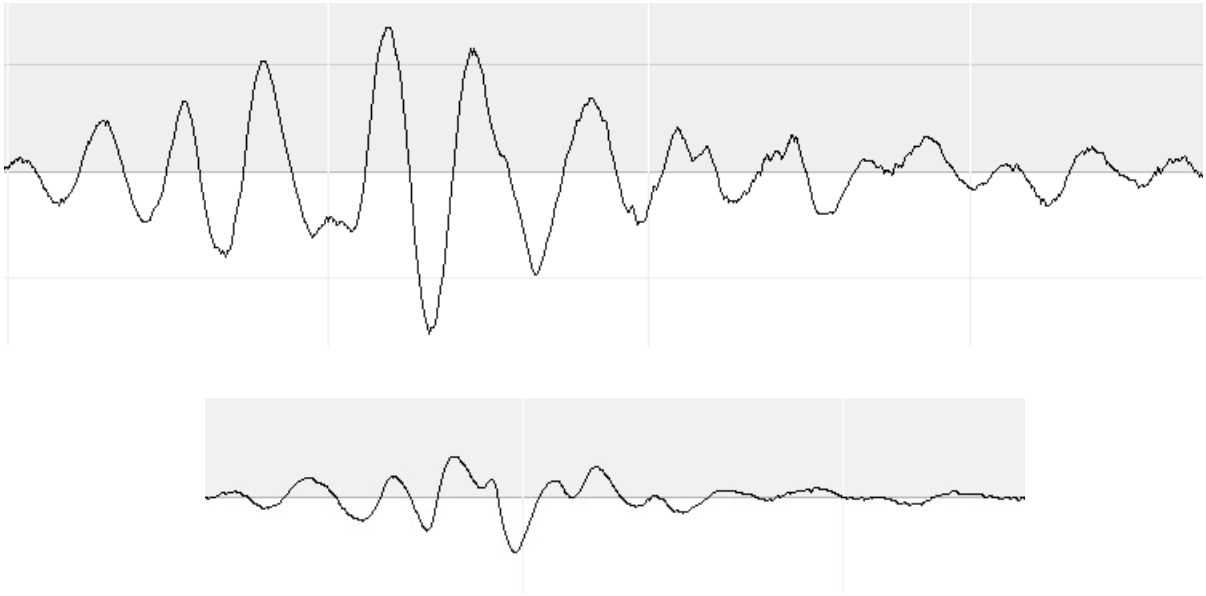
If you look at the signal, you see it has length 32465. Sampled at 8012Hz, it represents 32465/8012 = 4.0520469 seconds of sound. Within those four seconds, you have five full heart beats (which consist of two major sounds, hence ten peaks). So, the patient has a heart rate of some 74 beats per minute -- well within normal ranges.

However, this is about FFT, so we prefer to ask, what frequency is the whole beat? Of course, it's 5/4.05= 1.23 beats per second. If we're more generous, and consider ten cycles instead of five, we see the beats are at a frequency of 2.46 bps. Here's the FFT: where's the peak at 2.46?

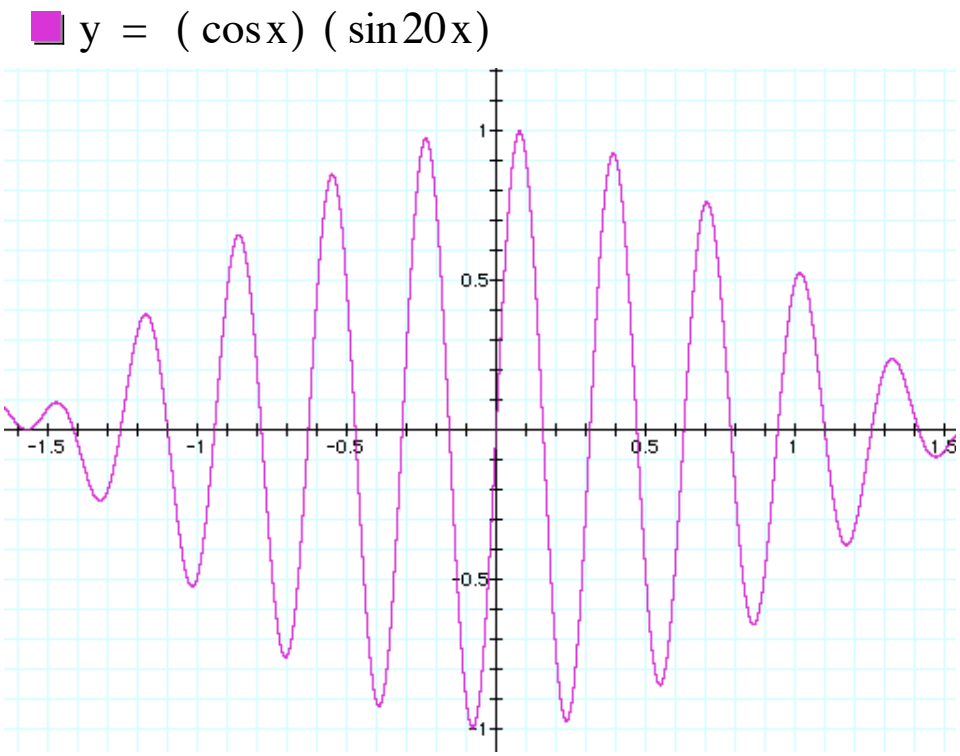


Ummm -- there isn't any. There's a grand old peak at around 125Hz, and a smaller one at 175Hz, but nothing for the main beat. What gives?

Let's look closer at a beat. Here's the two peaks in one beat, beat1.wav and beat2.wav.



Neither is a pure tone, what they remind me of most is a modulated signal:

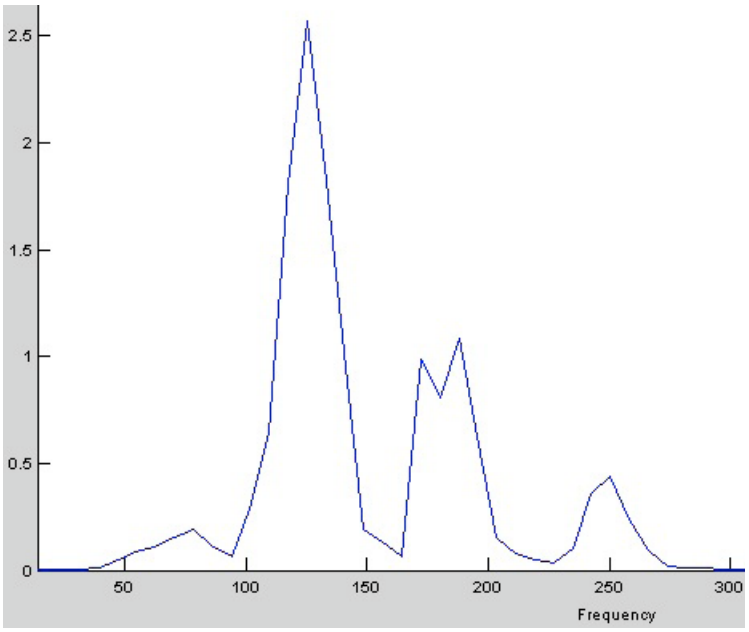


If I tried to pick out the modulating cosine in the above graph, I'd integrate y against the cosine. But the signal y oscillates up and down so quickly, the various positive and negative parts of the integral cancel, and I get no contribution from the cosine at all. The only frequencies that could possibly contribute to the FFT would be those close to sin(20x). Or, if we want to be rigorous instead of intuitinve, using trig identities:

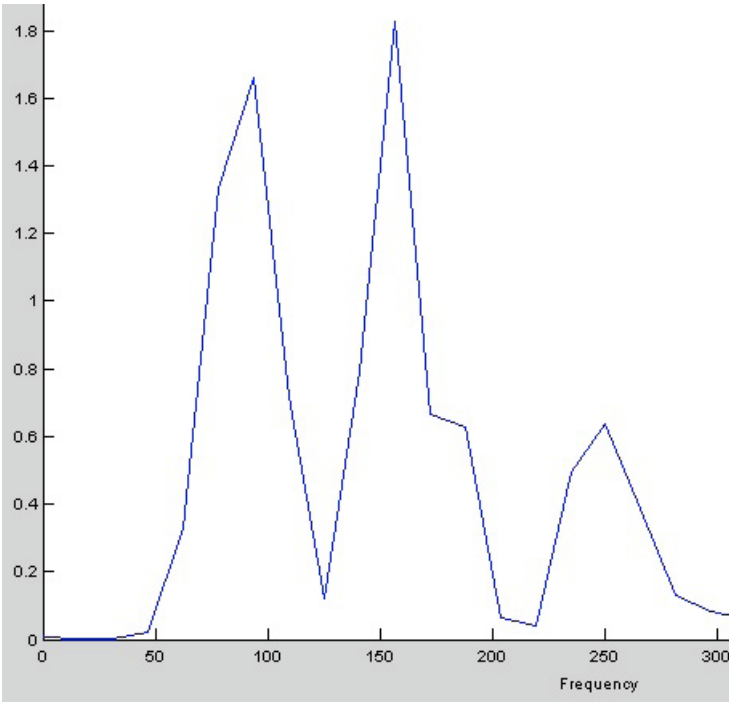
$$\cos(x)\sin(20x) = \frac{1}{2}\left[\sin(20+1)x + \sin(20-1)x\right]$$

The modulated sine wave has frequencies close to the original. This is why the heart beat appears to have no frequencies at 1.23 or 2.46Hz. Those are the modulating frequencies of some much higher sine.

Let's try using that approach. We can find the frequency of the sine that's being modulated in beat1 by counting the number of cycles in beat1: half the number of zero crossings. Beat1 has 14 cycles in 772 points, which gives me a sine of frequency 145.29Hz. Let's confirm that by taking the FFT of beat1:



Note the major peak between 100 and 150Hz; our computations aren't too far off. Similarly, beat2 has 9 cycles in 491 points, giving a frequency of 73.42Hz. And here's the FFT of beat2:



Again, a peak about where we'd expect it. Although let's not get too thrilled with ourselves; there are other peaks as well, suggesting we haven't got the full story of this FFT. But it's the kind of start that tells us we are beginning to understand what's going on in this signal.

Is there no hope, then, of recovering the modulating frequency? we'd have to find a way to eliminate the cancellations. And the simplest way to do that is to take the absolute value of the signal. Try it -- take the absolute value of beat.wav, then FFT, and inspect near 1Hz. What do you see?