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.

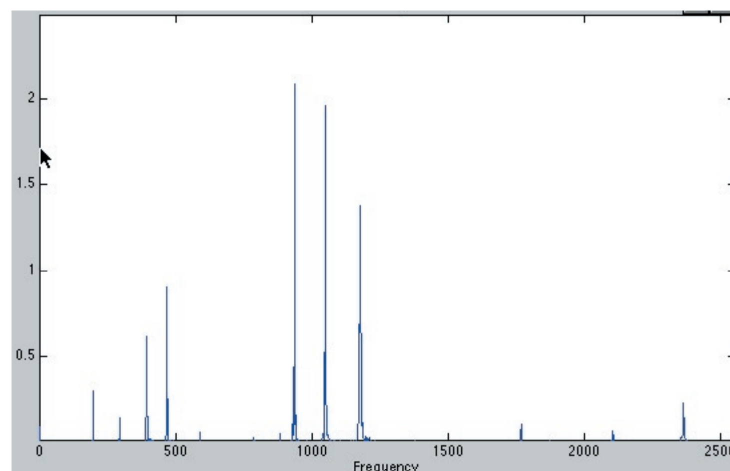
A quick computation before we use Matlab: let's assume we're taking the FFT of some music, from a CD. Every second, we have 44100 numbers, so a file lasting three minutes involves M = 7,938,000 numbers: music can eat up computer memory really fast.

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?

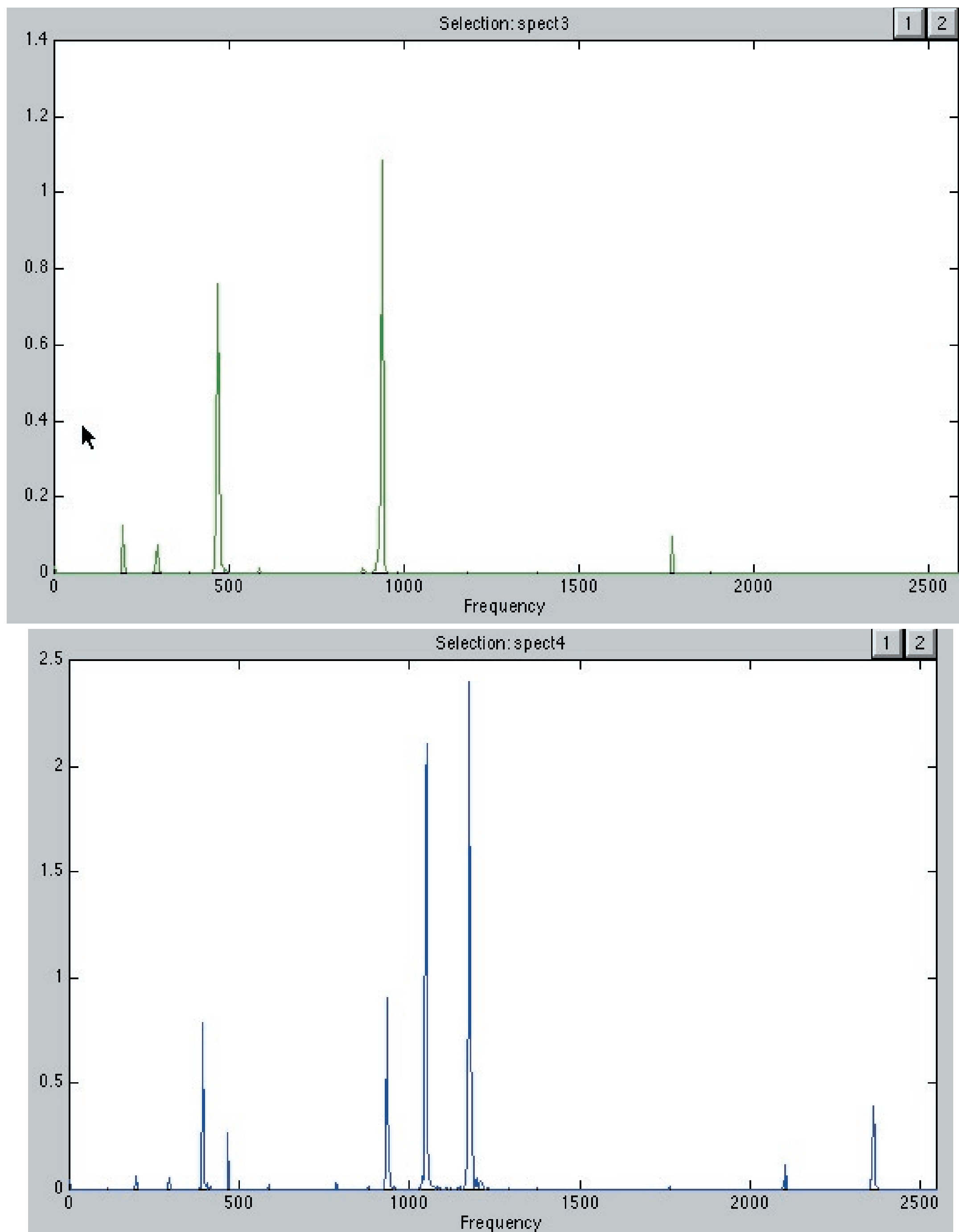
On the flip side (for those of us who grew up with vinyl records, which have flip sides), we might not care what the frequencies are for a whole recording. If we wanted to find when the guitar enters a piece, that's something that'll happen in a fraction of a second. Computing all the frequencies through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

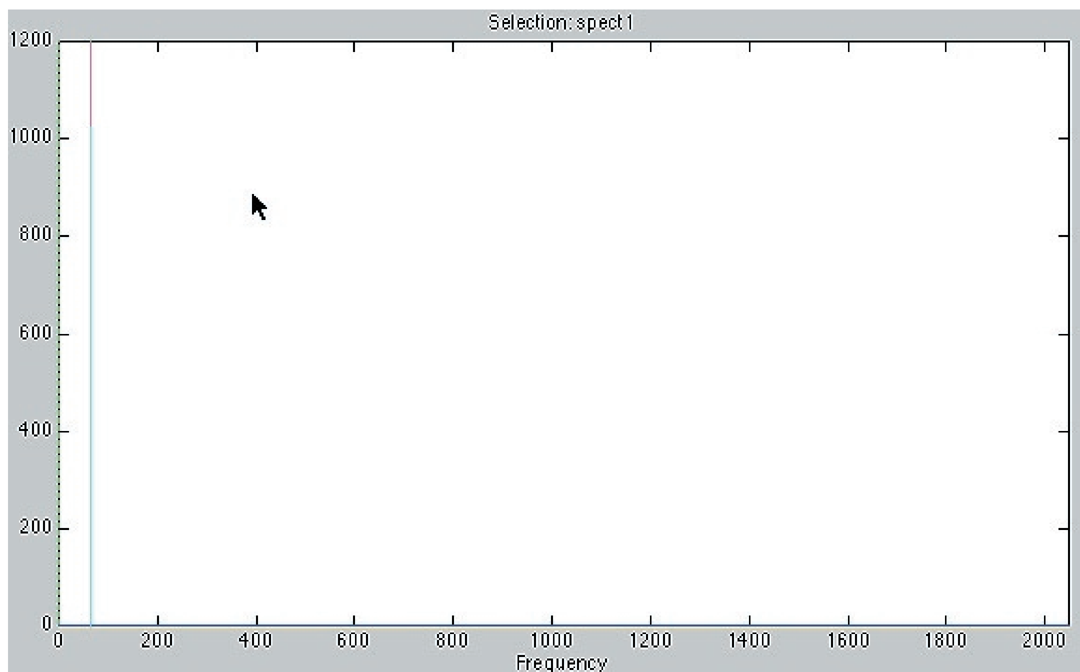
```
x=linspace(0,1, 4096);

%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.

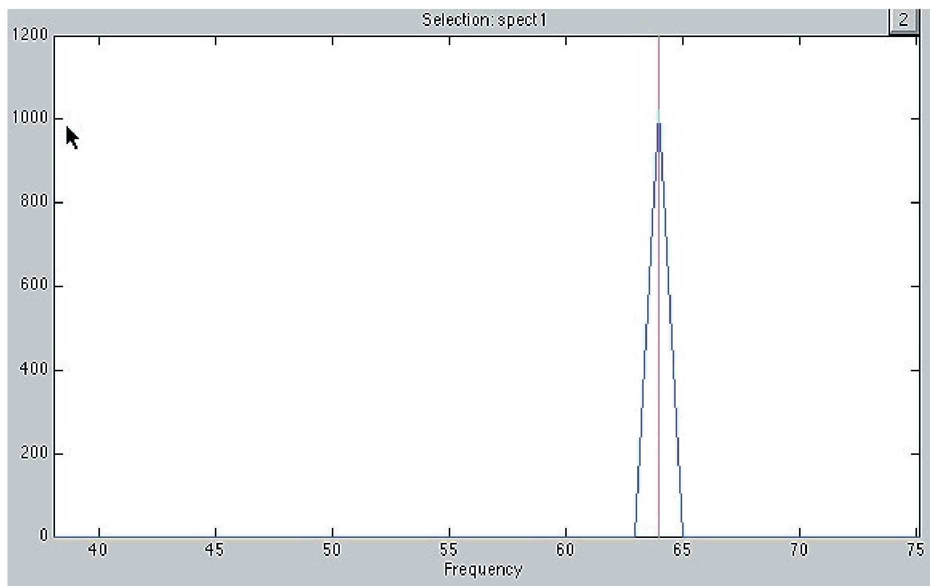
sptool offers a variety of ways to analyze the signal; start by using FFT: select it from the pull down menu. Your next choice is the number of points to analyze; since we have a signal with length 4096, take all 4096 points.

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:

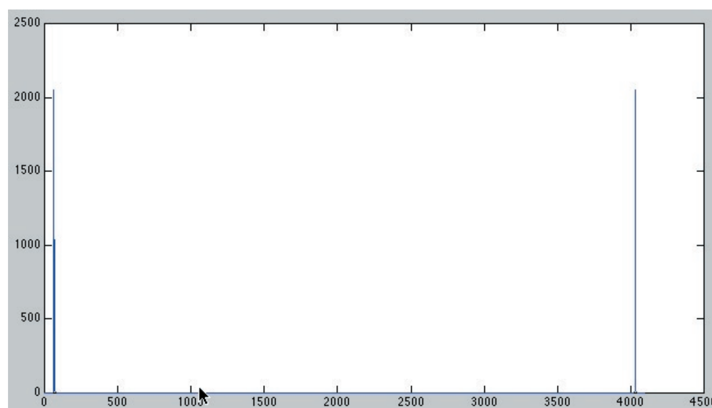


It looks nicer, in a way; certainly you can see the peak at 64 much more clearly. But there's a warning: the little triangle you see isn't really there. After all, the FFT only computes the transform at integer points, and every single point on the slopes of that triangle is at a non-integer point.

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

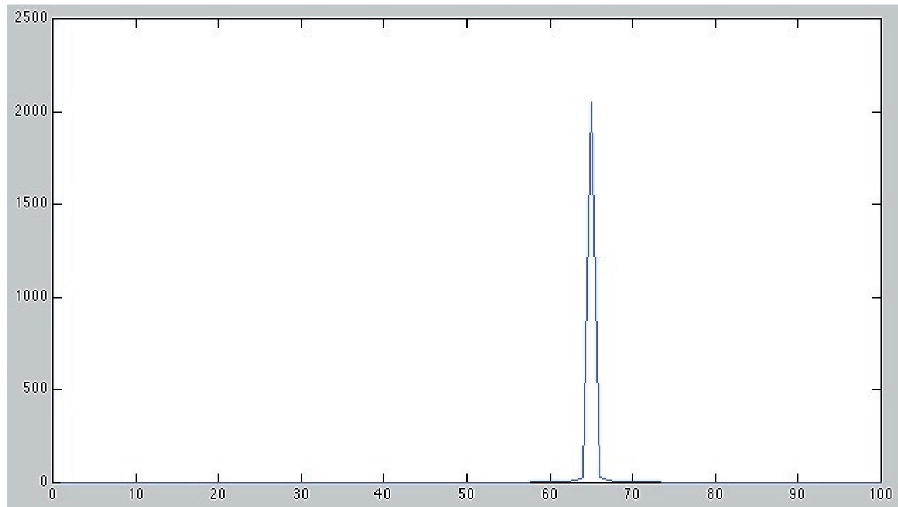
```
z64=fft(y64);
plot(abs(z64))
```

(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

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

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

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
```

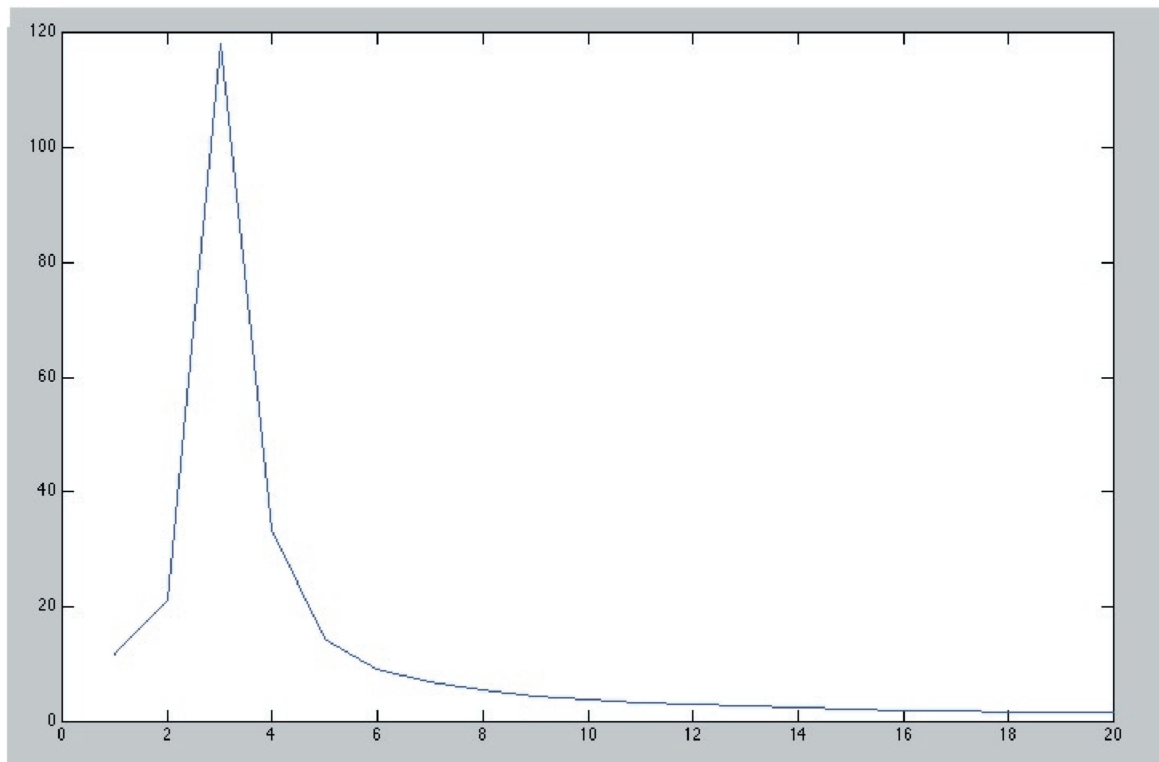
Which matches our expectations. The third entry is the transform evaluated at 64hz, and it ought to return N=128/2 (half because of the sine). It goes without saying we want to compute this by hand, using the definition of the FFT. Why miss a chance for fun? After all, the old computations don’t apply,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

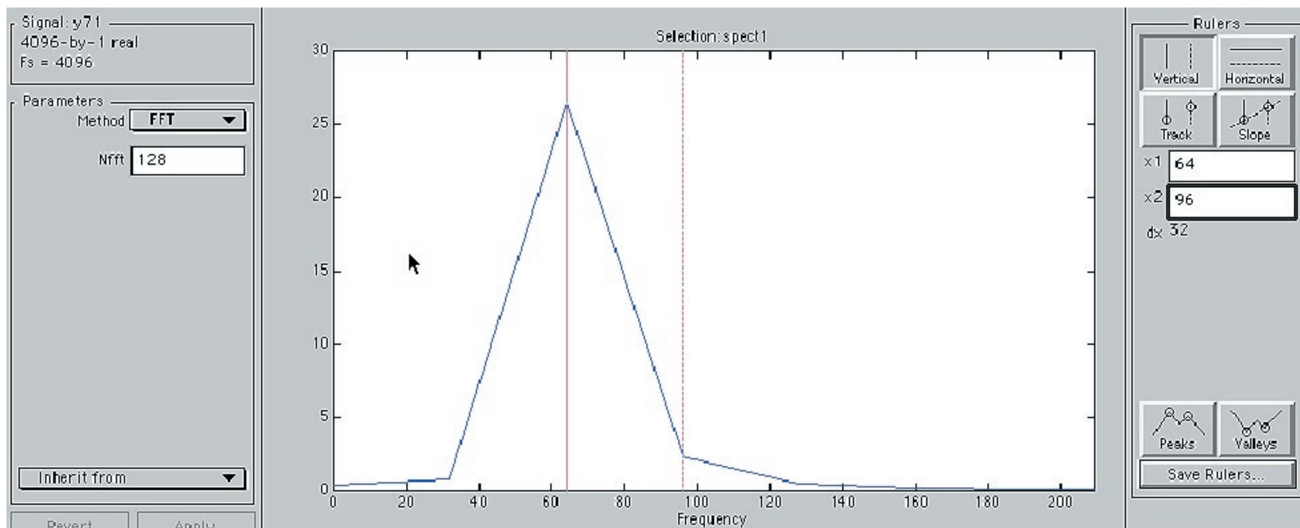
OK. Now the real test -- this time M=4096, N=128, as before, but we'll analyze y71. Here's what the FFT gives:



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 rewritten as

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

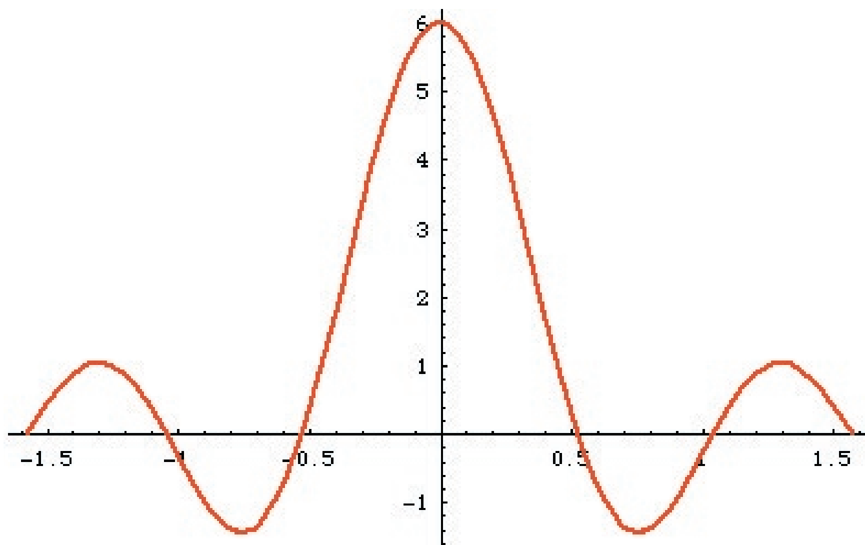
Since $a,\ b$ are imaginary, the difference is a sine . . .

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

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 $n=0,\ 1,\ ,\dots,\ 127$. 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 kth 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;

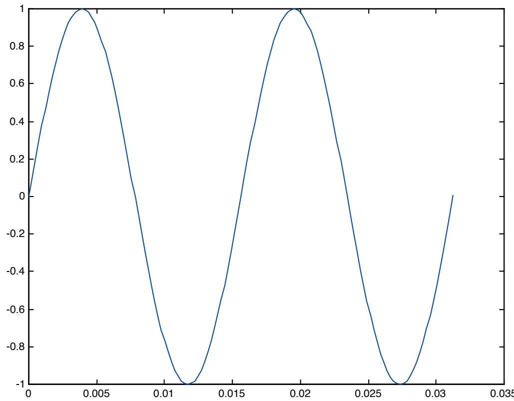
as above, y64 is periodic on N=128 points.

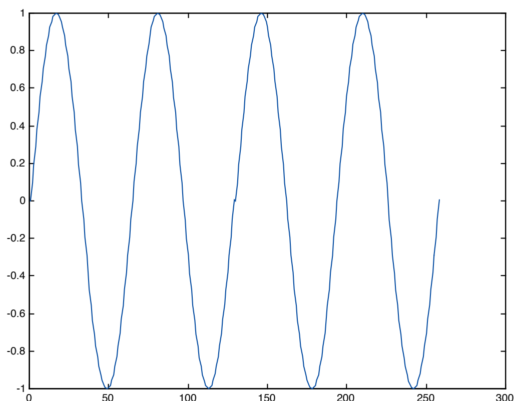
BUT, try it for y71 . . .
y71(129)=sin(2*pi*71*x(129)) = sin(2*pi*71*128/4096) = sin(2*pi*71/32) = .9807 . . .

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.

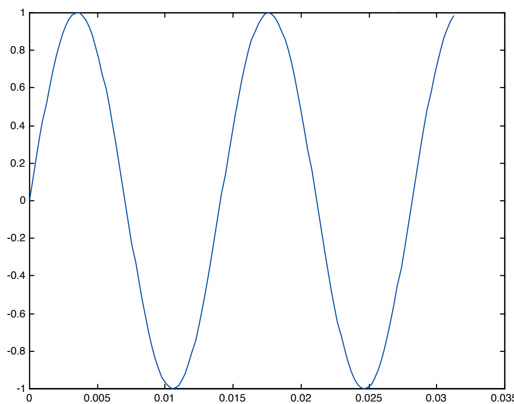
This is all very nice to know, but . . . . how does it explain Dirichlet kernels? Think of the FFT another way: it computes the Fourier transform of the periodic extension of y64 and y71. So let’s see what those periodic extensions look like.

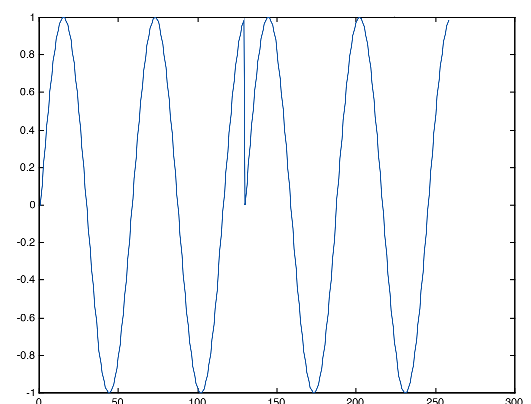
First, y64(k) for k=1, . . . 129:





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

computing the Fourier transform at $\theta[k] = \theta_0 + j\Delta\theta$ for $j = 0,\ 1,\ \ldots, K$. You want

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

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

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

Now plug back into the sum, to get

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

. If I define

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

then what we have is

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

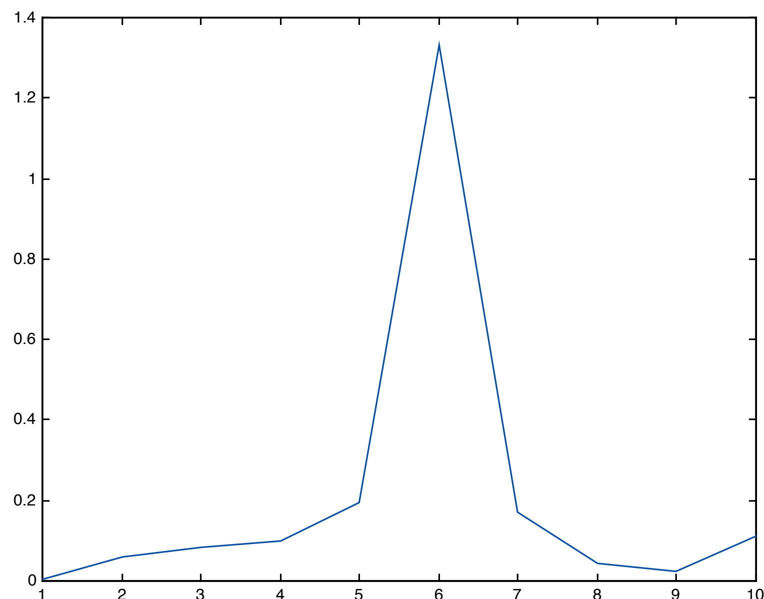
Up to a phase factor, this is a convolution. But convolutions and FFT's are easily related; $a \star b =$ IFFT [FFT [a]] · [FFT [b]].

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)

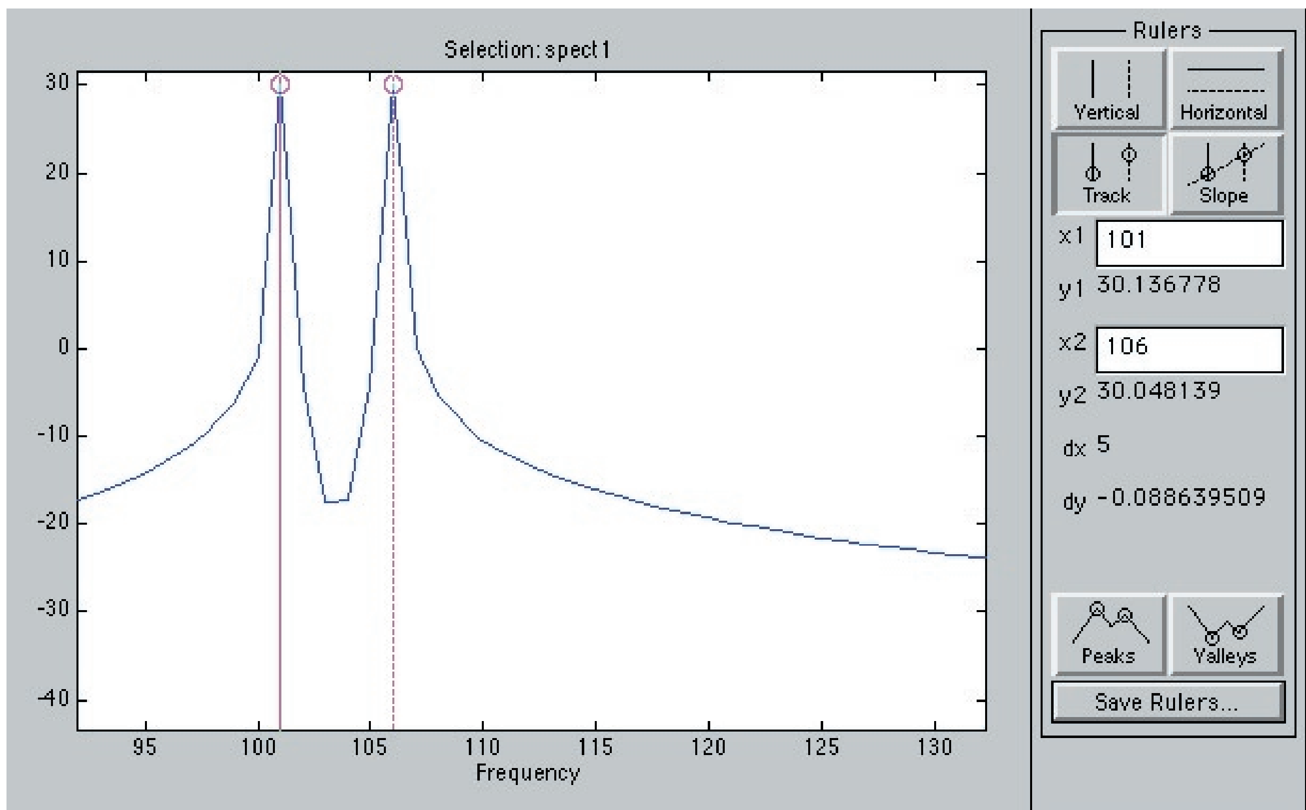
Chirp gives me the correct values, even when the FFT is flaking out.

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.
```

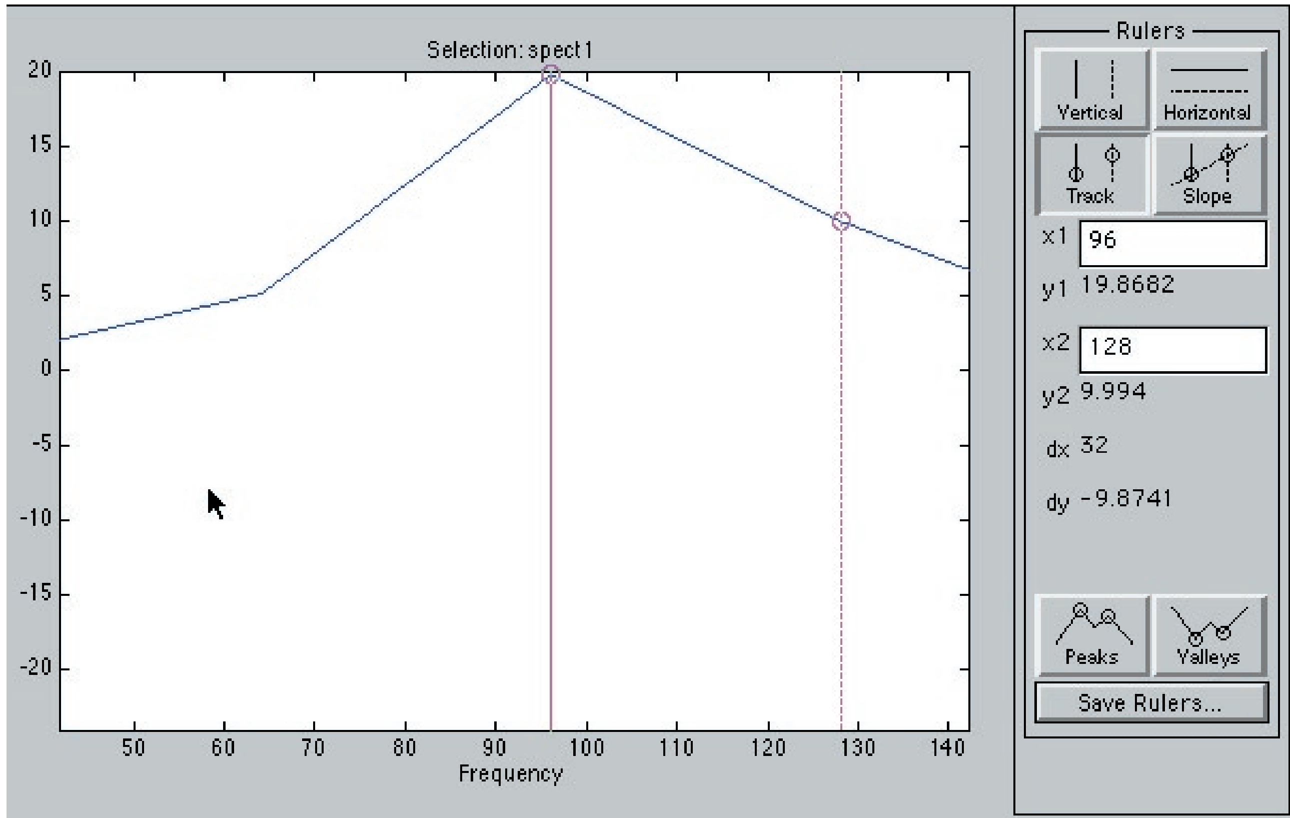
We'll leave behind chirp and friends, to give another perspective on the poor resolution of the FFT and the uncertainty principle; we want to see what the FFT does when it gets two frequencies. Let **y101_106 = sin(2*pi*101*x)+ sin(2*pi*106*x)**. Here it is, first at N=4096



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

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

In our case, $M/N = 32,\ n = 2$. Thus, frequencies ω with $32 \leq \omega \leq 96$ 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!