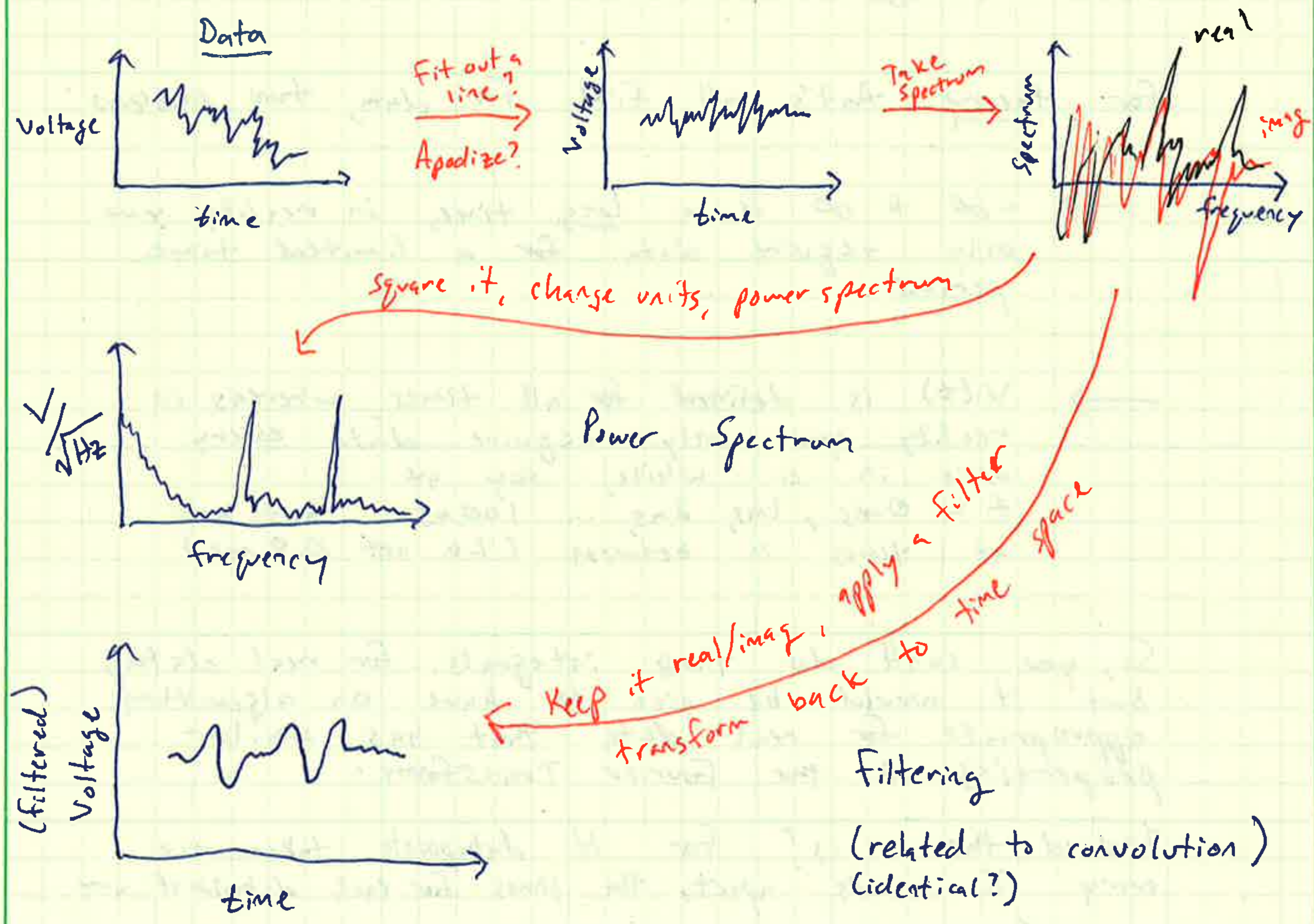
(Fast) Fourier Transform Notes

Sean Bryan
ASU

(Sources/Inspiration: Numerical Recipes in C
John Ruhl's Powerpoint Slides
Shaul Hanany's IDL code)

Often, converting a data analysis problem to spectral space is useful. In general, I picture this as:



These are the two problems we will talk about today. Keep in mind you can do this all with 2D images, or 3D volumes, but 1D is useful and I think an important place to start.

Math:

The Fourier transform, in <u>theory</u>, is defined as

$$\tilde{V}(f) = \int_{-\infty}^{\infty} V(t)\, e^{i 2\pi f t}\, dt$$

If you have $\tilde{V}(f)$ and you want to transform back, you have

$$V(t) = \int_{-\infty}^{\infty} \tilde{V}(f)\, e^{-i 2\pi f t}\, df$$

For <u>theory</u> that's all fine. For data, two problems:

→ $-\infty$ to ∞ is a <u>long</u> time, in reality you only acquire data for a limited time period

→ $V(t)$ is defined for all times, whereas in reality you only acquire data every once in a while, say at $t = 0\,ms, 1ms, 2ms, \ldots 100\,ms$ but not at times in between (i.e. not $0.5\,ms$)

So, you can't do those integrals for real data, but it would be nice to have an algorithm appropriate for real data that has similar properties to the Fourier Transform.

Indeed there is! For N datapoints taken once every τ seconds apart, the times for each datapoint are

$$t_k = \{ 0, \tau, 2\tau, 3\tau, \ldots, k\tau, \ldots (N-1)\tau \}$$

The voltage samples are

$$V_k = \{ V(0), V(\tau), \ldots V((N-1)\tau) \}$$

The Fourier Transform is then defined as

$$\tilde{V}_n = \sum_{k=0}^{N-1} V_k \, e^{i 2\pi n \frac{k}{N}}$$

time index

number of data points

frequency index

From before, the integral turned into a sum, time went from a continuous variable <u>with units</u> to just an integer counter k, and frequency went from a continuous variable <u>with units</u> to just an integer counter n.

Both V_k and $\tilde{V}_n$ still have <u>units</u>... but let's be <u>careful</u>!

The inverse transform, to go from $\tilde{V}_n$ back to V_k, shows some of the tricky issues with units.

$$V_k = \frac{1}{N} \sum_{n=0}^{N-1} \tilde{V}_n \, e^{-i 2\pi n \frac{k}{N}}$$

looks similar, minus sign is fine

who ordered <u>that</u>?

So, different numerical codes that implement these equations put the N in different places, but ... it's always lurking no matter what.

Also watch the minus signs... grr... and the N

	Numerical Recipes	Matlab	IDL	Python
$\tilde{V}_n =$	$\sum V_k e^{i 2\pi n \frac{k}{N}}$	$\sum V_k e^{-i 2\pi n \frac{k}{N}}$	$\frac{1}{N} \sum V_k e^{-i 2\pi n \frac{k}{N}}$	$\sum V_k e^{-i 2\pi n \frac{k}{N}}$
$V_k =$	$\frac{1}{N} \sum \tilde{V}_n e^{-i 2\pi n \frac{k}{N}}$	$\frac{1}{N} \sum \tilde{V}_n e^{i 2\pi n \frac{k}{N}}$	$\sum \tilde{V}_n e^{i 2\pi n \frac{k}{N}}$	$\frac{1}{N} \sum \tilde{V}_n e^{i 2\pi n \frac{k}{N}}$

So... um... I'll consider python, and note that it's the same in matlab, the same as NR up to a minus sign, and be <u>very</u> careful of the N in IDL.

Putting the "n" in $\tilde{V}_n$ ... what are these frequencies?

Thinking back to the original integrals:

$$V(t) = \int_{-\infty}^{\infty} \tilde{V}(f)\, e^{-i 2\pi f t}\, df$$

For a general complex-valued function there are both <u>positive</u> and <u>negative</u> frequencies.

For the fft, if we use the fftshift function, this becomes clearly the case too.

$$yfft = fftshift\Big(\cancel{fft}\,(\,ydata\,) \Big)$$

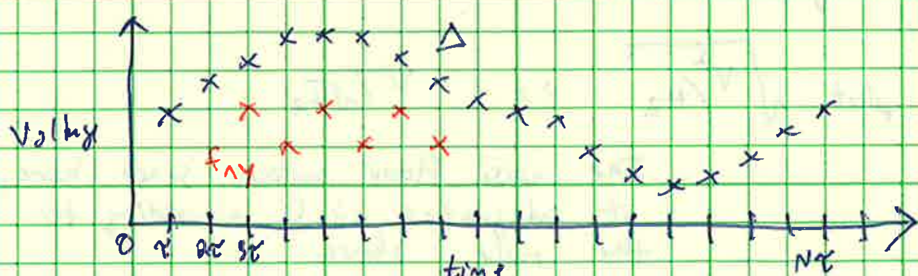
Here, the companion frequency array is

$\Delta = 1/N\tau \qquad f_{ny} = 1/2\tau$

$$\text{Frequency array} = \left\{ -f_{ny},\ -f_{ny}+\Delta,\ -f_{ny}+2\Delta,\ \ldots,\ 0, \Delta, \ldots f_{ny}-\Delta \right\}$$

Δ represents the lowest possible signal frequency you could see in your entire dataset. It oscillates once in the dataset.

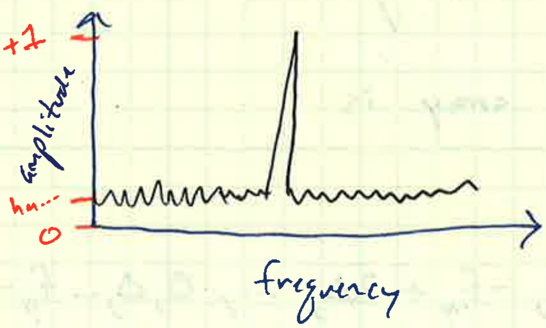
f_{ny} is the Nyquist frequency and is the highest possible signal frequency you could see given how often you are sampling the data. It oscillates up in one sample then down in the next.



Normalization Conventions: Power/Bandwidth or Amplitude?

Consider a signal which is a sine wave and broadband white noise

Setting aside implementation, two possible kinds of plots we might want exist. One is a plot of amplitude:



This graph the peak is +1, great! Easy to interpret. But... the noise level is weird. Since we normalized based on sine wave amplitude... what's a "sine wave amplitude" of noise? Like, what does that mean?

Another would be a power/bandwidth kind of plot. Noise power adds incoherently so units of V^2/Hz make sense because

$$\text{rms in volts} = \sqrt{\int_{\text{bandwidth}} \frac{V^2}{Hz}\, df}$$

We can plot $\sqrt{V^2/Hz}$ i.e. $V/\sqrt{Hz}$

The noise floor makes sense here, it integrates nicely according to the rule above.

But... my sine wave isn't noise...?

Achieving these Normalizations

For a real valued function, the positive and negative frequency bins of the FFT contain nearly identical information

$$V(t) \text{ real} \implies \tilde{V}(-f) = \text{complex conjugate}\left(\tilde{V}(f)\right)$$

So, we ignore the negative frequencies, and multiply the positive frequency $\tilde{V}$ values by two. This "conserves information" in some sense. N real numbers for the input voltage time-ordered-data, and $N/2$ complex numbers for the output voltage FFT.

The zero frequency and f_{ny} bins already have some subtle redundancy, so don't multiply them by two.

With that... let's go to the python code and try out all this background theory!

Aside:

Matlab uses FFTW, "The fastest FFT in the West"

It's fast! ← i.e. 33 million data points

`data = randn(2^25,1); tic; tmp = fft(data); toc;`

Elapsed time: 1 second winner!

Numpy seems like a good place to start, but it both doesn't use FFTW and does not check if the input is real.

Elapsed time: 3 seconds

Scipy fftpack doesn't use FFTW, but at least it checks to see if the input is real

Elapsed time: 1.5 seconds Best compromise?

In principle, there exists pyFFTW, but it wouldn't install on windows.

Elapsed time: In theory fast, but won't install

(Read up on scipy.fftpack.rfft if you want... not to much faster on my machine, maybe 1.3 seconds?)