GEOPHYSICAL TUTORIAL, June 2018

Time-frequency decomposition

by Matt Hall, Agile Scientific, matt@agilescientific.com

Consider a waveform or signal s as a function of time t. For example, a sine wave with some amplitude a and at some frequency f might be defined by:

$$s(t) = a \sin(2\pi f t)$$

We can implement this mathematical function as a subroutine, usually also called a *function*, in the Python programming language. Since computers live in a discrete world, we'll need to evaluate the function over some duration and at some sample rate:

```
def sine_wave(f, a, duration, sample_rate):
    t = np.arange(0, duration, 1/sample_rate)
    return a * np.sin(2 * np.pi * f * t), t
```

We can now call this function, passing it a frequency $f = 261.63$ Hz. We'll ask for 0.25 seconds, with a sample rate of 10 kHz.

```
s, t = sine_wave(f=261.63,
                 a=1,
                 duration=0.25,
                 sample_rate=10e3)
```

This results in the following signal, commonly called a *time series*, which we visualize by plotting s against time t:

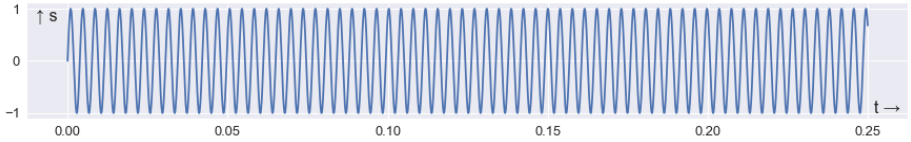

Figure 1: png

I've plotted the resulting array as a line, but it's really a series of discrete points represented in Python as an array of numbers, beginning with these four:

```
array([ 0.        ,  0.1636476 ,  0.32288289,  0.47341253])
```

Let's plot the first 80 points:

When air vibrates at this frequency, we hear a middle C, or C4. You can hear the note for yourself in the Jupyter Notebook accompanying this article at

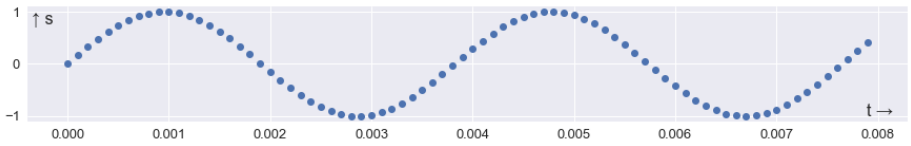

Figure 2: png

https://github.com/seg/tutorials-2018 (the notebook also contains all the code for making the plots). The code to render the array `s` as audio is very short:

```
from IPython.display import Audio
fs = 10e3
Audio(s, rate=fs)
```

This signal is only 0.25 seconds long and there are already a lot of wiggles. We'd love to have seismic at this frequency! Most seismic data is only played on the lower 20 to 30 keys of an 88-key piano — indeed the lowest key is A0, which at 27.5 Hz is above the peak frequency of many older surveys.

If we wanted to know the frequency of this signal, we could assume that it's a pure tone and simply count the number of cycles per unit time. But natural signals are rarely monotones, so let's make something more complicated. We can use our function to make the C-major chord with 3 notes, C4, E4, and G4 by passing column vectors (by reshaping the arrays) for frequency and amplitude:

```
f = np.array([261.6, 329.6, 392.0])
a = np.array([1.5, 0.5, 1])
s, t = sine_wave(f=f.reshape(3, 1),
                 a=a.reshape(3, 1),
                 duration=0.25,
                 sample_rate=10e3)
```

The result is a set of three sine curves 0.25 seconds long:

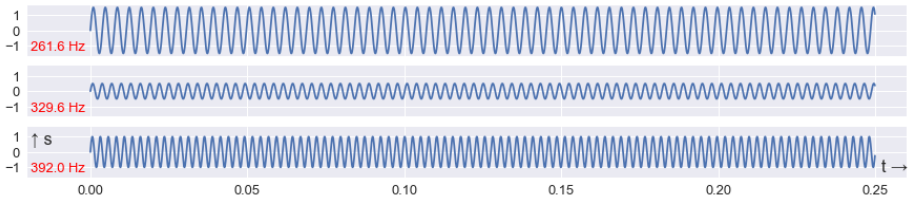

Figure 3: png

The total signal is given by the sum of the three curves:

```
s = np.sum(s, axis=0)
```

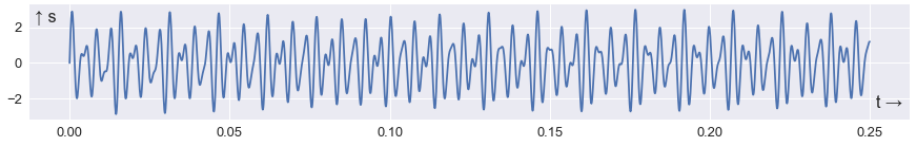

Figure 4: png

The Fourier transform

Although this mixed or *polytonic* signal is just the sum of three pure tones, it is no longer a trivial matter to figure out the components. This is where the Fourier transform comes in.

We won't go into how the Fourier transform works — for what it's worth, the best explanation I've seen recently is the introductory video by Grant Sanderson (3Blue1Brown on YouTube). The point is that the transform describes signals as mixtures of periodic components. Let's try it out on our chord.

First we *taper* the signal by multiplying it by a *window* function. Ideal pure tones have infinite duration, and the tapering helps prevent the edges of the signal from interfering with the Fourier transform.

```
s = s * np.blackman(s.size)
```

The window function (green) has a tapering effect on the signal:

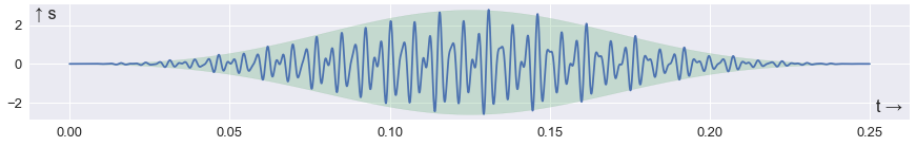

Figure 5: png

Because the function s is defined for a given moment in time t, we call this representation of the signal the time domain.

NumPy's fast Fourier transform function `fft()` takes the signal $s(t)$ and returns a new representation of the signal, $S(f)$ (sometimes also called $\hat{s}(f)$. This new representation is called the frequency domain. It consists of an array of *Fourier coefficients*:

```
S = np.fft.fft(s)
```

A helper function, `fftfreq()`, returns the array of frequencies corresponding to the coefficients. The frequency sample interval is determined by the duration of the signal s: the longer the signal, the smaller the frequency sample interval.

(Similarly, short sample intervals in time correspond to broad bandwidth in frequency.)

```
freq = np.fft.fftfreq(s.size, d=1/10e3)
```

The result is an array of *Fourier coefficients*, most of which are zero. But at and near the frequencies in the chord, the coefficients are large. The result: a 'recipe' for the chord, in terms of sinusoidal monotones.

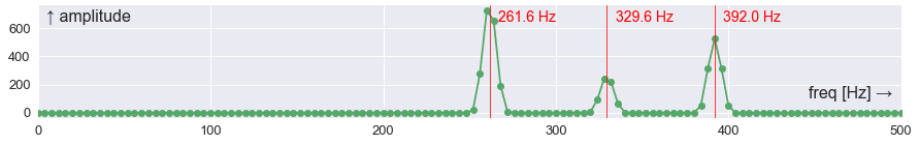

Figure 6: png

This is called the *spectrum* of the signal *s*. It shows the magnitude of each frequency component.

Time-frequency representation

We now know how to unweave polytonic signals, but let's introduce another complication — signals whose components change over time. Such signals are said to be *nonstationary*. For example, think of a monotonic signal whose tone changes at some moment (see the Notebook for the code that generates this signal):

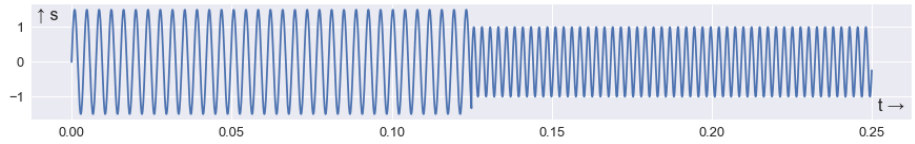

Figure 7: png

We can compute the Fourier transform of this signal, just as before:

```
s *= np.blackman(s.size)
S = np.fft.fft(s)
freq = np.fft.fftfreq(s.size, d=1/10e3)
```

And plot amplitude `S` against the frequency array `freq`:

It looks very similar to the spectrum we made before, but without the middle frequency. The peaks are a bit more spread out because the duration of each waveform is half what it was (the general uncertainty principle spreads signals out in frequency as they become more compact in time).

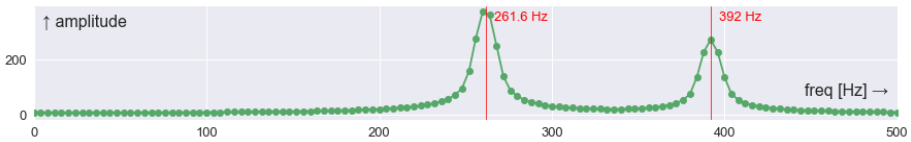

Figure 8: png

The point is that there's not much difference between the spectrum of two mixed signals, and the spectrum of two consecutive signals. If we care about the localization of signals in time (we do!), this is a problem. One solution is to turn to *time–frequency representations*. by attempting to break the signal down in time and frequency simultaneously, they offer a way to enjoy the advantages of both domains at the same time.

Python's `matplotlib` plotting library offers a convenient way of making a time–frequency plot, also known as a *spectrogram*. In a single line of code, it produces a 2D image plot showing frequency against time.

```
_ = plt.specgram(s, Fs=1/10e3, NFFT=512, noverlap=480)
```

With a bit more work, we can make a very rich view of our data:

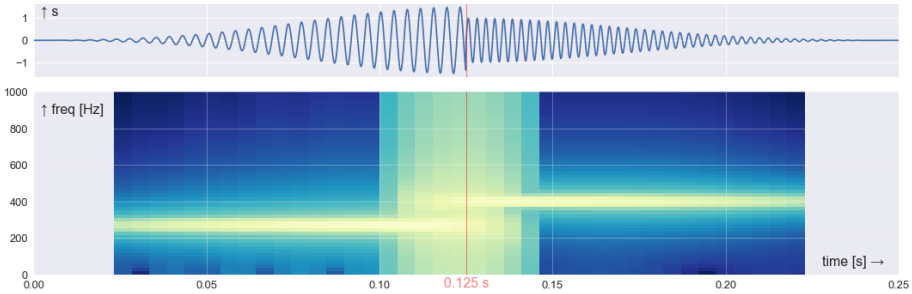

Figure 9: png

The plot uses an algorithm called the short-time Fourier transform, or STFT. This simply makes a Fourier transform in a sliding window of length `NFFT`, with `noverlap` points overlapping on the previous window. We want `NFFT` to be long to get good frequency resolution, and we want `noverlap` to be large to get good time resolution.

Notice that we cannot quite see the exact frequency of the components — they don't last long enough to pin them down. And there's a bit of uncertainty about the timing of the transition, because to get decent frequency resolution we need a longish segment of the signal (512 samples in this case) — so we lose timing information. But overall, this plot is an improvement over the spectrum alone: we can see that there are at least 2 strong signals, with frequencies of about 250

and 400 hertz.

A piece of piano music might resemble this kind of plot. Because piano keys can only play one note, piano music looks like a series of horizontal lines:

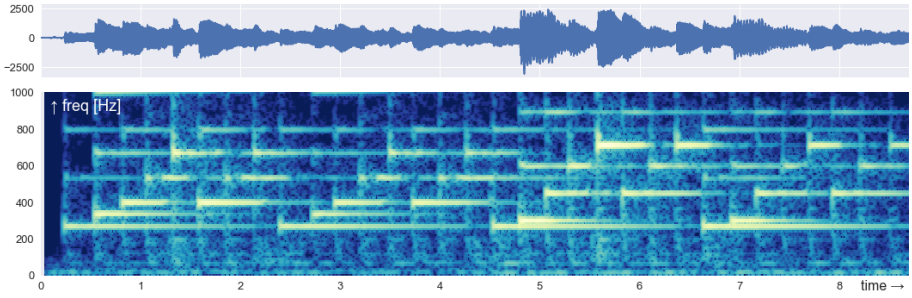

Figure 10: png

There is a strong similarity between this time–frequency decomposition and the musical staff notation:

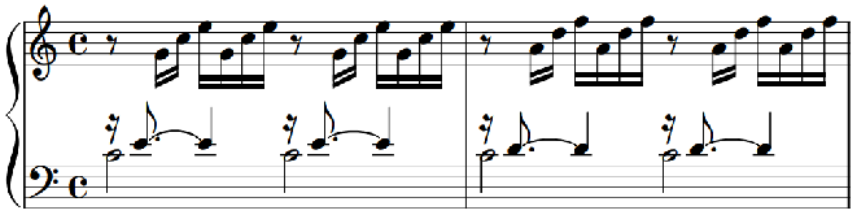

Figure 11: png

It turns out that most interesting signals — and perhaps all natural signals — are polytonic and nonstationary. For this reason, while the timeseries is often useful, a time–frequency decomposition can be very revealing. Here are some examples; in each case, frequency is on the vertical axis and time is on the horizontal axis. The colours indicate low (blue) to high (yellow) power (proportional to the square of the amplitude).

Here's a human voice saying, "SEG". The sonorant vowel sounds have harmonics (horizontal stripes), while the sibilant sounds of the "S" and the first part of the "G" have noise-like spectral responses.

This spectrogram shows a 5-second series of bat chirps. I've indicated 18 kHz, the approximate limit of adult human hearing, with a red line, and if you listen to the audio of this signal in the Notebook, you can verify that the chirps are barely audible at normal playback speed; only by slowing the clip down can they be clearly heard.

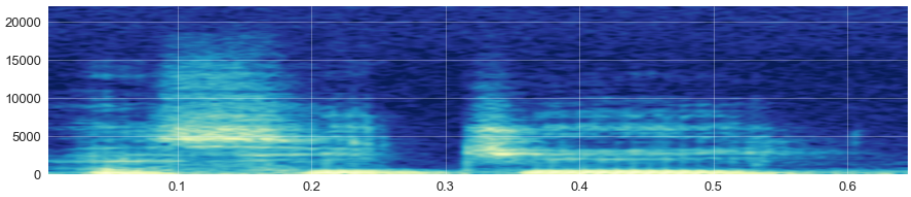

Figure 12: png

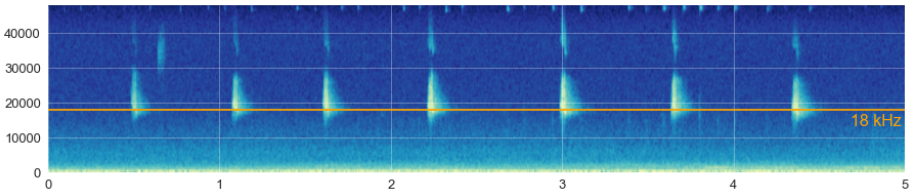

Figure 13: png

Finally, here's a volcanic 'scream' — a harmonic tremor preceeding an explosive eruption at Mt Redoubt, Alaska, in March 2009. It sounds incredible in audio, but the spectrogram is interesting too. In contrast to the bat chirp, this 15-minute-long time series has to be sped up in order to hear it.

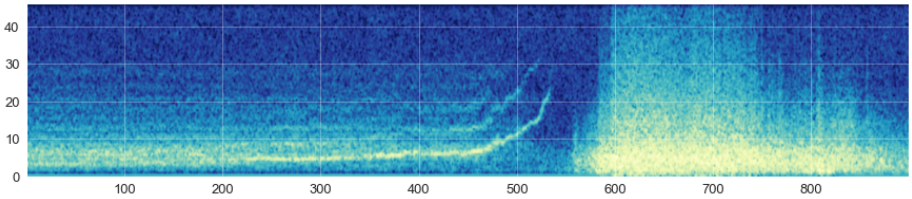

Figure 14: png

Continue exploring

All of the figures in this notebook can be reproduced by the code in the Jupyter Notebook accompanying this article at https://github.com/seg/tutorials-2018. You can even run the code on the cloud and play with it in your browser. You can't break anything — don't worry!

You'll also find more signals in the repository, synthetic and natural, from heartbeats and mysterious underwater chirps to gravitational waves and seismic traces. Not only that, there's a notebook showing you how to use another algorithm — the continuous wavelet transform — to get a different kind of

time–frequency decomposition.

Happy decomposition!

Acknowledgments

The piano recording from BWV846 by Bach is licensed CC-BY by Kimiko Ishizaka on welltemperedclavier.org. The bat chirp data is licensed CC-BY-NC by freesound.org user klankschap. Thank you to Alicia Hotovec-Ellis for her help with the Mt Redoubt data, recorded by the USGS Alaska Volcano Observatory.