Chapter 4. Frequency and the Fast Fourier Transform

If you want to find the secrets of the universe, think in terms of energy, frequency and vibration.

—Nikola Tesla

This chapter was written in collaboration with SW's father, PW van der Walt.

This chapter will depart slightly from the format of the rest of the book. In particular, you may find the *code* in the chapter quite modest. Instead, we want to illustrate an elegant *algorithm*, the Fast Fourier Transform (FFT), that is endlessly useful, is implemented in SciPy, and works, of course, on NumPy arrays.

Introducing Frequency

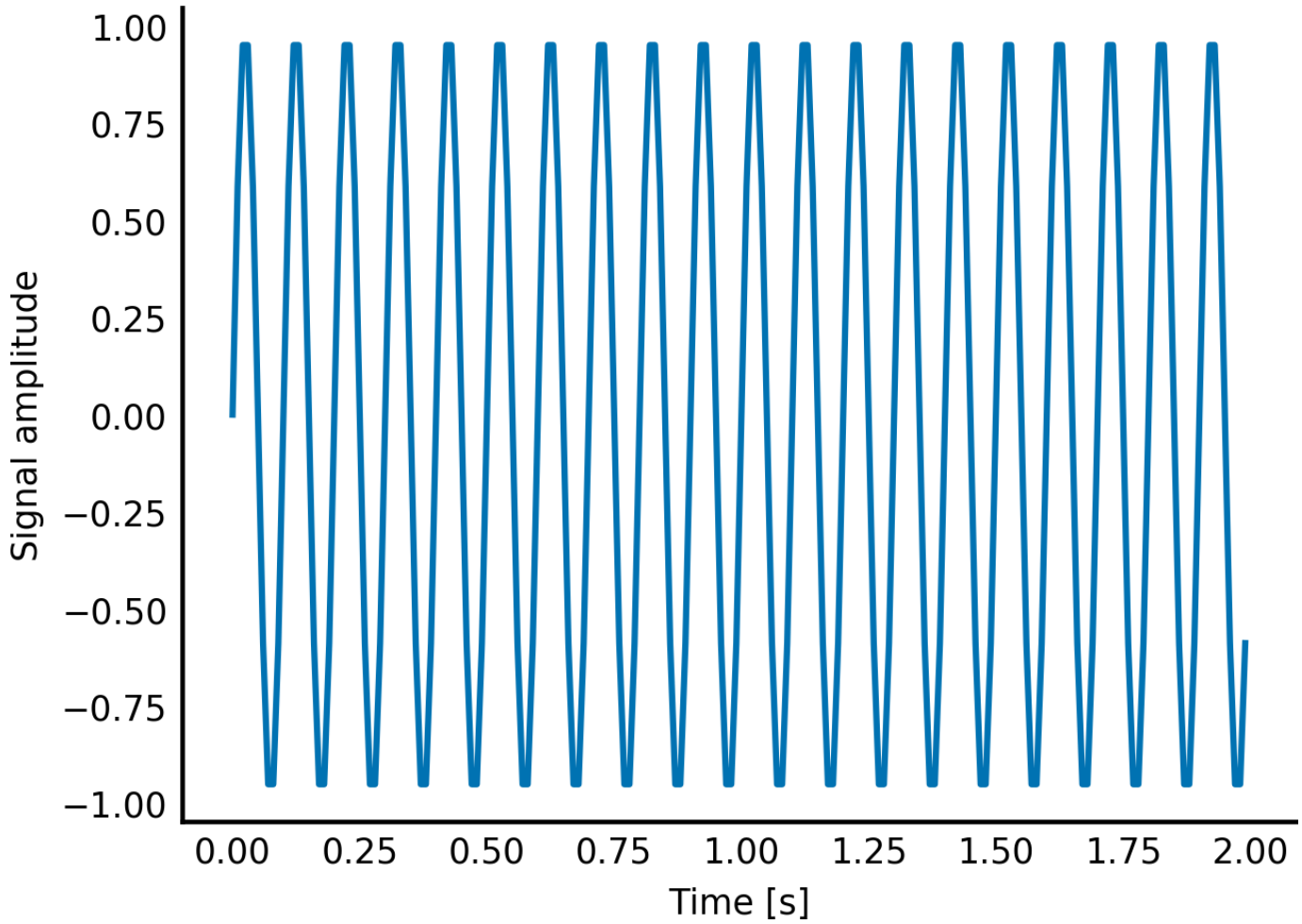
We'll start by setting up some plotting styles and importing the usual suspects:

```
# Make plots appear inline, set custom plotting style
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

```
import numpy as np
```

The discrete[1] Fourier transform (DFT) is a mathematical technique used to convert temporal or spatial data into *frequency domain* data. *Frequency* is a familiar concept, due to its colloquial occurrence in the English language: the lowest notes your headphones can rumble out are around 20 Hz, whereas middle C on a piano lies around 261.6 Hz; Hertz, or oscillations per second, in this case literally refers to the number of times per second at which the membrane inside the headphone moves to-and-fro. That, in turn, creates compressed pulses of air which, upon arrival at your eardrum, induces a vibration at the same frequency. So, if you take a simple periodic function, $sin(10 \times 2\pi t)$, you can view it as a wave:

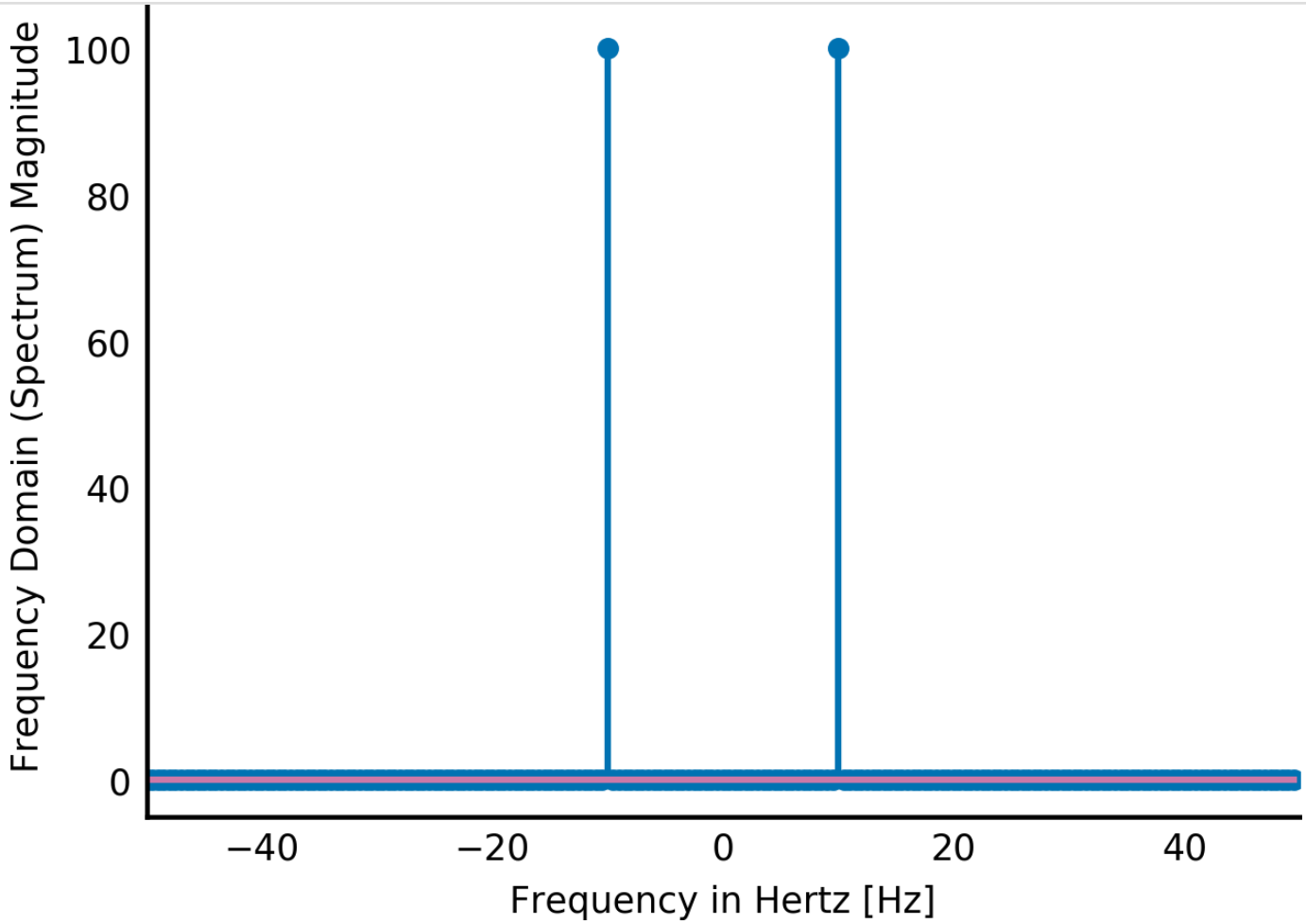
```
f = 10  # Frequency, in cycles per second, or Hertz
f_s = 100  # Sampling rate, or number of measurements per second
```



Or you can equivalently think of it as a repeating signal of *frequency* 10 Hz (it repeats once every 1/10 seconds—a length of time we call its *period*). Although we naturally associate frequency with time, it can equally well be applied to space. For example, a photo of a textile patterns exhibits high *spatial frequency*, whereas the sky or other smooth objects have low spatial frequency.

Let us now examine our sinusoid through application of the DFT:

```
from scipy import fftpack

X = fftpack.fft(x)
freqs = fftpack.fftfreq(len(x)) * f_s

fig, ax = plt.subplots()

ax.stem(freqs, np.abs(X))
ax.set_xlabel('Frequency in Hertz [Hz]')
ax.set_ylabel('Frequency Domain (Spectrum) Magnitude')
```



We see that the output of the FFT is a 1D array of the same shape as the input, containing complex values. All values are zero, except for two entries. Traditionally, we visualize the magnitude of the result as a *stem plot*, in which the height of each stem corresponds to the underlying value.

(We explain why you see positive and negative frequencies later on in "Discrete Fourier Transforms". You may also refer to that section for a more in-depth overview of the underlying mathematics.)

The Fourier transform takes us from the *time* to the *frequency* domain, and this turns out to have a massive number of applications. The *fast Fourier transform* (FFT) is an algorithm for computing the DFT; it achieves its high speed by storing and reusing results of computations as it progresses.

In this chapter, we examine a few applications of the DFT to demonstrate that the FFT can be applied to multidimensional data (not just 1D measurements) to achieve a variety of goals.

Illustration: A Birdsong Spectrogram

Let's start with one of the most common applications, converting a sound signal (consisting of variations of air pressure over time) to a *spectrogram*. You might have seen spectrograms on your music player's

Figure 4-1. The Numark EQ2600 stereo equalizer (image used with permission from the author, Sergey Gerasimuk)

Listen to this snippet of the nightingale birdsong (released under CC BY 4.0):

```
from IPython.display import Audio
Audio('data/nightingale.wav')
```

If you are reading the paper version of this book, you'll have to use your imagination! It goes something like this: chee-chee-woorrrr-hee-hee cheet-wheet-hoorrr-chirrr-whi-wheo-wheo-wheo-wheo-wheo-wheo.

Since we realize that not everyone is fluent in bird-speak, perhaps it's best if we visualize the measurements—better known as "the signal"—instead.

We load the audio file, which gives us the sampling rate (number of measurements per second) as well as audio data as an `(N, 2)` array—two columns because this is a stereo recording.

```
audio = np.mean(audio, axis=1)
```

Then, we calculate the length of the snippet and plot the audio (Figure 4-2).

```
N = audio.shape[0]
L = N / rate

print(f'Audio length: {L:.2f} seconds')

f, ax = plt.subplots()
ax.plot(np.arange(N) / rate, audio)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Amplitude [unknown]');
```

```
Audio length: 7.67 seconds
```

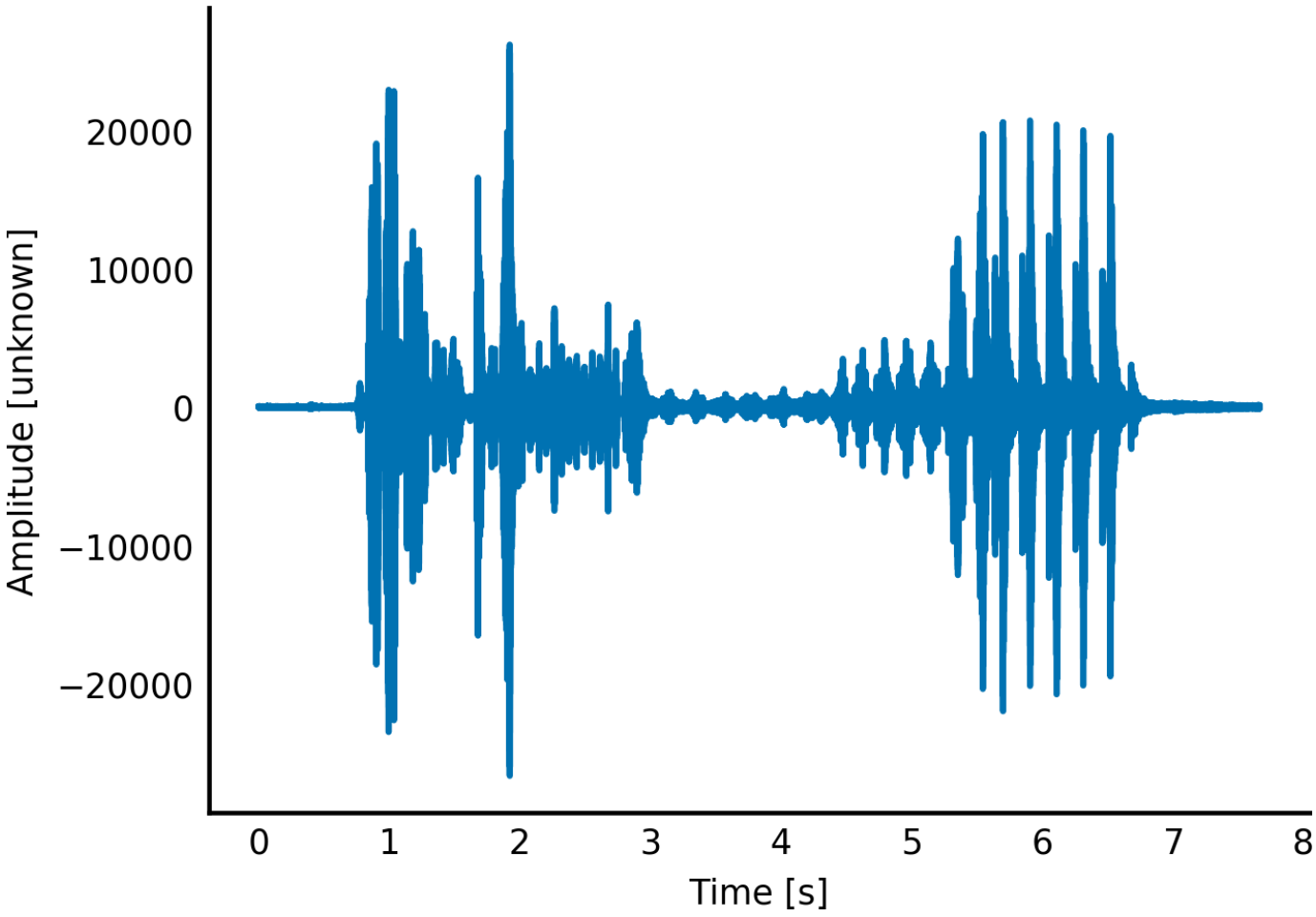


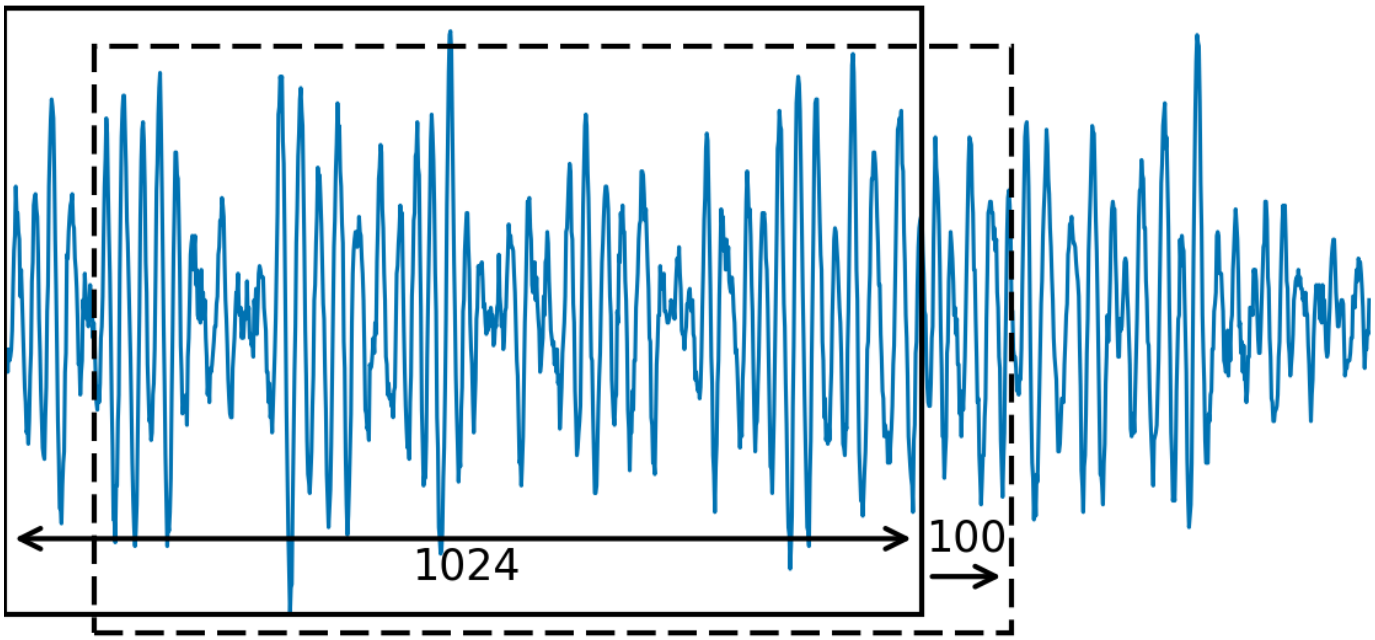
Figure 4-2. Audio waveform plot of nightingale birdsong

used to refer to the DFT itself. The DFT tells us which frequencies or "notes" to expect in our signal.

Of course, a bird sings many notes throughout the song, so we'd also like to know *when* each note occurs. The Fourier transform takes a signal in the time domain (i.e., a set of measurements over time) and turns it into a spectrum—a set of frequencies with corresponding (complex[2]) values. The spectrum does not contain any information about time![3]

So, to find both the frequencies and the time at which they were sung, we'll need to be somewhat clever. Our strategy is as follows: take the audio signal, split it into small, overlapping slices, and apply the Fourier transform to each (a technique known as the *short time Fourier transform*).

We'll split the signal into slices of 1,024 samples—that's about 0.02 seconds of audio. The reason we chose 1,024 and not 1,000 we'll explain in a second when we examine performance. The slices will overlap by 100 samples as shown here:



Start by chopping up the signal into slices of 1024 samples, each slice overlapping the previous by 100 samples. The resulting `slices` object contains one slice per row.

```
from skimage import util

M = 1024

slices = util.view_as_windows(audio, window_shape=(M,), step=100)
print(f'Audio shape: {audio.shape}, Sliced audio shape: {slices.shape}')
```

```
win = np.hanning(M + 1)[:-1]
slices = slices * win
```

It's more convenient to have one slice per column, so we take the transpose:

```
slices = slices.T
print('Shape of `slices`:', slices.shape)
```

```
Shape of `slices`: (1024, 3371)
```

For each slice, calculate the DFT, which returns both positive and negative frequencies (more on that in "Frequencies and Their Ordering"), so we slice out the positive M2 frequencies for now.

```
spectrum = np.fft.fft(slices, axis=0)[:M // 2 + 1:-1]
spectrum = np.abs(spectrum)
```

(As a quick aside, you'll note that we use `scipy.fftpack.fft` and `np.fft` interchangeably. NumPy provides basic FFT functionality, which SciPy extends further, but both include an `fft` function, based on the Fortran FFTPACK.)

The spectrum can contain both very large and very small values. Taking the log compresses the range significantly.

Here we do a log plot of the ratio of the signal divided by the maximum signal (shown in Figure 4-3). The specific unit used for the ratio is the decibel, $20log_{10}$ (amplitude ratio).

```
f, ax = plt.subplots(figsize=(4.8, 2.4))

S = np.abs(spectrum)
S = 20 * np.log10(S / np.max(S))

ax.imshow(S, origin='lower', cmap='viridis',
          extent=(0, L, 0, rate / 2 / 1000))
ax.axis('tight')
ax.set_ylabel('Frequency [kHz]')
ax.set_xlabel('Time [s]');
```

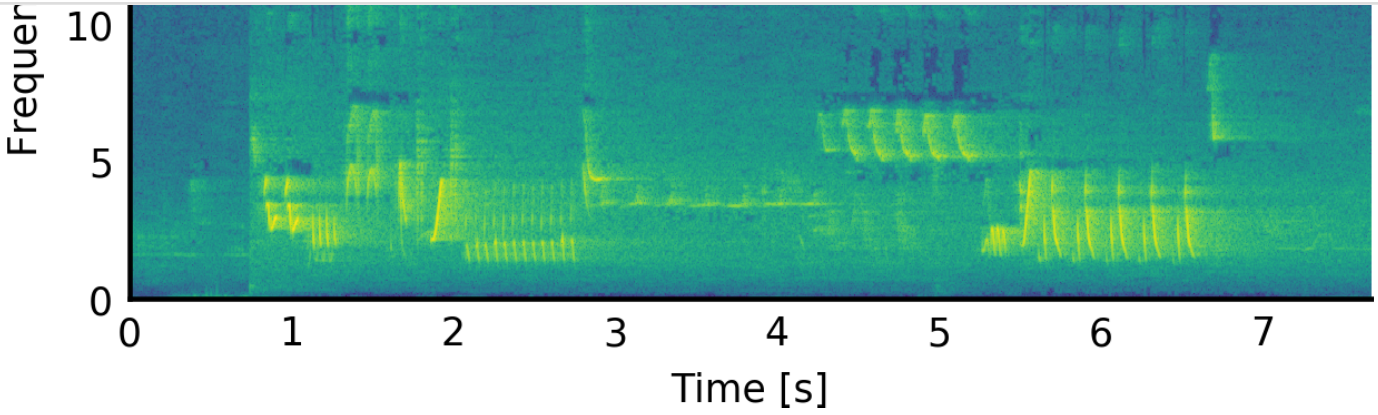


Figure 4-3. Birdsong spectrogram

Much better! We can now see that the frequencies vary over time, and the spectrogram corresponds to the way the audio sounds. See if you can match our earlier description: chee-chee-woorrrr-hee-hee cheet-wheet-hoorrr-chirrr-whi-wheo-wheo-wheo-wheo-wheo-wheo. (I didn't transcribe the 3 to 5 second mark—that's another bird.)

SciPy already includes an implementation of this procedure as `scipy.signal.spectrogram` (Figure 4-4), which can be invoked as follows:

```
from scipy import signal

freqs, times, Sx = signal.spectrogram(audio, fs=rate, window='hanning',
                                      nperseg=1024, noverlap=M - 100,
                                      detrend=False, scaling='spectrum')

f, ax = plt.subplots(figsize=(4.8, 2.4))
ax.pcolormesh(times, freqs / 1000, 10 * np.log10(Sx), cmap='viridis')
ax.set_ylabel('Frequency [kHz]')
ax.set_xlabel('Time [s]');
```

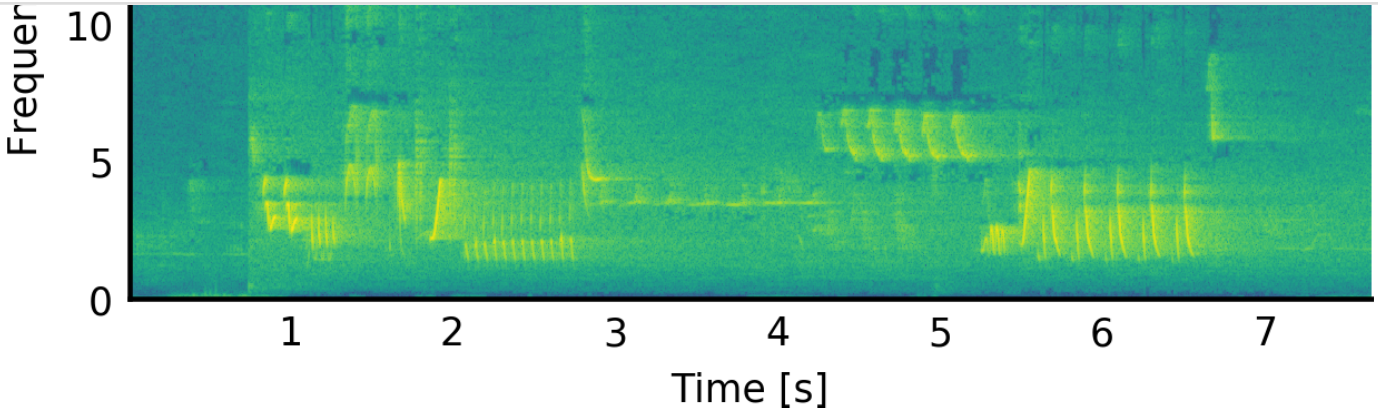


Figure 4-4. SciPy built-in rendition of birdsong spectrogram

The only differences between the manual spectrogram that we created versus the SciPy's built-in function are that SciPy returns the spectrum magnitude squared (which turns measured voltage into measured energy), and multiplies it by some normalization factors.[4]

History

Tracing the exact origins of the Fourier transform is tricky. Some related procedures go as far back as Babylonian times, but it was the hot topics of calculating asteroid orbits and solving the heat (flow) equation that led to several breakthroughs in the early 1800s. Whom exactly among Clairaut, Lagrange, Euler, Gauss, and D'Alembert we should thank is not exactly clear, but Gauss was the first to describe the fast Fourier transform (an algorithm for computing the DFT, popularized by Cooley and Tukey in 1965). Joseph Fourier, after whom the transform is named, first claimed that *arbitrary* periodic[5] functions can be expressed as a sum of trigonometric functions.

Implementation

The DFT functionality in SciPy lives in the `scipy.fftpack` module. Among other things, it provides the following DFT-related functionality:

`fft`, `fft2`, `fftn`

Compute the DFT using the FFT algorithm in 1, 2, or `n` dimensions.

`ifft`, `ifft2`, `ifftn`

Compute the inverse of the DFT.

`dct`, `idct`, `dst`, `idst`

`fftfreq`

Return the DFT sample frequencies.

`rfft`

Compute the DFT of a real sequence, exploiting the symmetry of the resulting spectrum for increased performance. Also used by `fft` internally when applicable.

This list is complemented by the following functions in NumPy:

`np.hanning`, `np.hamming`, `np.bartlett`, `np.blackman`, `np.kaiser`

Tapered windowing functions.

The DFT is also used to perform fast convolutions of large inputs by `scipy.signal.fftconvolve`.

SciPy wraps the Fortran FFTPACK library—it is not the fastest out there, but unlike packages such as FFTW, it has a permissive free software license.

Choosing the Length of the DFT

A naive calculation of the DFT takes $O(N^2)$ operations.[6] How come? Well, you have N (complex) sinusoids of different frequencies ($2\pi f \times 0$, $2\pi f \times 1$; $2\pi f \times 3$, ..., $2\pi f \times (N - 1)$), and you want to see how strongly your signal corresponds to each. Starting with the first, you take the dot-product with the signal (which, in itself, entails N multiplication operations). Repeating this operation N times, once for each sinusoid, then gives N^2 operations.

Now, contrast that with the FFT, which is $O(N \log N)$ in the ideal case due to the clever reuse of calculations—a great improvement! However, the classical Cooley-Tukey algorithm implemented in FFTPACK (and used by SciPy) recursively breaks up the transform into smaller (prime-sized) pieces and only shows this improvement for "smooth" input lengths (an input length is considered smooth when its largest prime factor is small, as shown in Figure 4-5). For large prime-sized pieces, the Bluestein or Rader algorithms can be used in conjunction with the Cooley-Tukey algorithm, but this optimization is not implemented in FFTPACK.[7]

Let us illustrate:

```
import time
```

```
smoothness = [max(factorint(i).keys()) for i in lengths]


exec_times = []
for i in lengths:
    z = np.random.random(i)

    # For each input length i, execute the FFT K times
    # and store the execution time

    times = []
    for k in range(K):
        tic = time.monotonic()
        fftpack.fft(z)
        toc = time.monotonic()
        times.append(toc - tic)

    # For each input length, remember the *minimum* execution time
    exec_times.append(min(times))


f, (ax0, ax1) = plt.subplots(2, 1, sharex=True)
ax0.stem(lengths, np.array(exec_times) * 10**6)
ax0.set_ylabel('Execution time (µs)')

ax1.stem(lengths, smoothness)
ax1.set_ylabel('Smoothness of input length\n(lower is better)')
ax1.set_xlabel('Length of input');
```

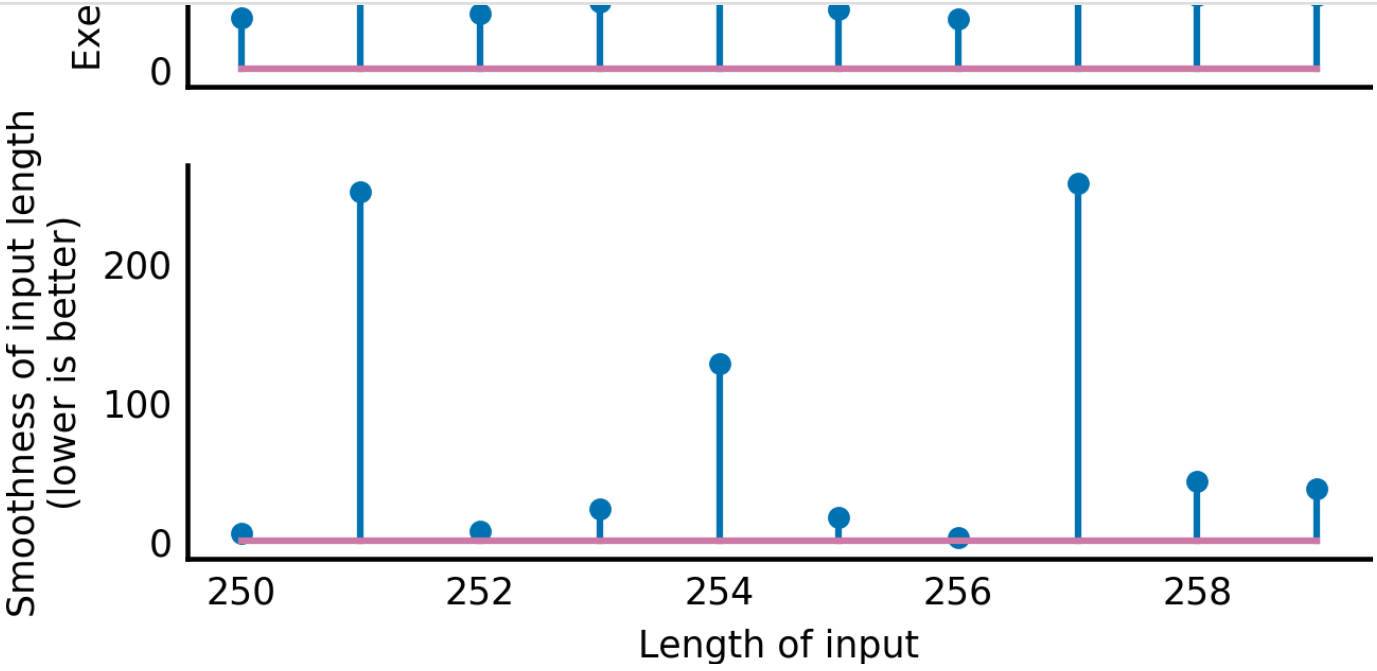


Figure 4-5. FFT execution time versus smoothness for different input lengths

The intuition is that, for smooth numbers, the FFT can be broken up into many small pieces. After performing the FFT on the first piece, we can reuse those results in subsequent computations. This explains why we chose a length of 1,024 for our audio slices earlier—it has a smoothness of only 2, resulting in the optimal "radix-2 Cooley-Tukey" algorithm, which computes the FFT using only $(N/2)\log_2 N = 5{,}120$ complex multiplications, instead of $N^2 = 1{,}048{,}576$. Choosing $N = 2^m$ always ensures a maximally smooth N (and, thus, the fastest FFT).

More DFT Concepts

Next, we present a couple of common concepts worth knowing before operating heavy Fourier transform machinery, whereafter we tackle another real-world problem: analyzing target detection in radar data.

Frequencies and Their Ordering

For historical reasons, most implementations return an array where frequencies vary from low to high to low (see "Discrete Fourier Transforms" for further explanation of frequencies). For example, when we do the real Fourier transform of a signal of all ones, an input that has no variation and therefore only has the slowest, constant Fourier component (also known as the "DC," or Direct Current, component—just electronics jargon for "mean of the signal"), appearing as the first entry:

```
from scipy import fftpack
N = 10
```

```
z = np.ones(10)
z[::2] = -1

print(f'Applying FFT to {z}')
fftpack.fft(z)
```

```
Applying FFT to [-1.  1. -1.  1. -1.  1. -1.  1. -1.  1.]
```

```
array([  0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j, -10.+0.j,
         0.-0.j,   0.-0.j,   0.-0.j,   0.-0.j])
```

Note that the FFT returns a complex spectrum that, in the case of real inputs, is conjugate symmetrical (i.e., symmetric in the real part and antisymmetric in the imaginary part):

```
x = np.array([1, 5, 12, 7, 3, 0, 4, 3, 2, 8])
X = fftpack.fft(x)

np.set_printoptions(precision=2)

print("Real part:     ", X.real)
print("Imaginary part:", X.imag)

np.set_printoptions()
```

```
Real part:      [ 45.     7.09 -12.24  -4.09  -7.76  -1.    -7.76  -4.09 -12.24
  7.09]
Imaginary part: [  0.   -10.96  -1.62  12.03   6.88   0.    -6.88 -12.03   1.62
 10.96]
```

(And, again, recall that the first component is `np.mean(x) * N`.)

The `fftfreq` function tells us which frequencies we are looking at specifically:

```
fftpack.fftfreq(10)
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4, -0.5, -0.4, -0.3, -0.2, -0.1])
```

resample the spectrum using the `resample` function.

With the numbers X_k known, the inverse DFT *exactly* recovers the sample values x_n through the following summation:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j2\pi kn/N}$$

Keeping in mind that $e^{j\theta} = cos\theta + j\ sin\ \theta$, the last equation shows that the DFT has decomposed the sequence x_n into a complex discrete Fourier series with coefficients X_k. Comparing the DFT with a continuous complex Fourier series:

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t}$$

The DFT is a *finite* series with N terms defined at the equally spaced discrete instances of the *angle* $(\omega_0 t_n) = 2\pi\frac{k}{N}$ in the interval [0, 2π)—that is, *including* 0 and *excluding* 2π. This automatically normalizes the DFT so that time does not appear explicitly in the forward or inverse transform.

If the original function $x(t)$ is limited in frequency to less than half of the sampling frequency (the so-called *Nyquist frequency*), interpolation between sample values produced by the inverse DFT will usually give a faithful reconstruction of $x(t)$. If $x(t)$ is *not* limited as such, the inverse DFT can, in general, not be used to reconstruct $x(t)$ by interpolation. Note that this limit does not imply that there are *no* methods that can do such a reconstruction—see, for example, compressed sensing, or finite rate of innovation sampling.

The function $e^{j2\pi k/N} = (e^{j2\pi/N})^k = w^k$ takes on discrete values between 0 and $2\pi\frac{N-1}{N}$ on the unit circle in the complex plane. The function $e^{j2\pi kn/N} = w^{kn}$ encircles the origin $n\frac{N-1}{N}$ times, thus generating harmonics of the fundamental sinusoid for which $n = 1$.

The way in which we defined the DFT leads to a few subtleties when $n > \frac{N}{2}$, for even N.[8] The function $e^{j2\pi kn/N}$ is plotted for increasing values of k in Figure 4-6, for the cases $n = 1$ to $n = N - 1$ for $N = 16$. When k increases from k to $k + 1$, the angle increases by $\frac{2\pi n}{N}$. When $n = 1$, the step is $\frac{2\pi}{N}$. When $n = N - 1$, the angle increases by $2\pi\frac{N-1}{N} = 2\pi - \frac{2\pi}{N}$. Since 2$\pi$ is precisely once around the circle, the step equates to $-\frac{2\pi}{N}$—that is, in the direction of a negative frequency. The components up to $N/2$ represent *positive* frequency components, those above $N/2$ up to $N - 1$ represent *negative* frequencies. The angle increment for the component $N/2$ for N even advances precisely halfway around the circle for each increment in k and can therefore be interpreted as either a positive or a negative frequency. This component of the DFT represents the Nyquist frequency (i.e., half of the sampling frequency), and is useful to orientate oneself when looking at DFT graphics.

The FFT in turn is simply a special and highly efficient algorithm for calculating the DFT. Whereas a straightforward calculation of the DFT takes of the order of N^2 calculations to compute, the FFT algorithm requires of the order N log N calculations. The FFT was key to the widespread use of DFT in real-time applications and was included in a list of the top 10 algorithms of the twentieth century by the *IEEE journal Computing in Science & Engineering* in 2000.

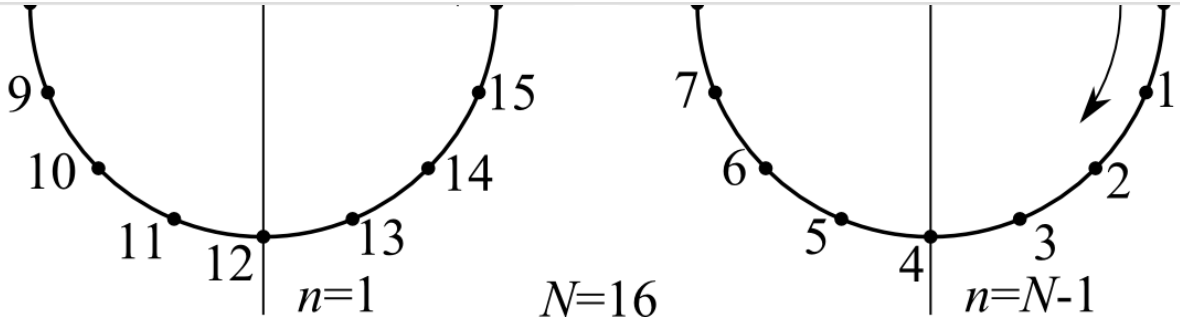


Figure 4-6. Unit circle samples

Let's examine the frequency components in a noisy image (Figure 4-7). Note that, while a static image has no time-varying component, its values do vary across *space*. The DFT applies equally to either case.

First, load and display the image:

```
from skimage import io
image = io.imread('images/moonlanding.png')
M, N = image.shape

f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(image)

print((M, N), image.dtype)
```

```
(474, 630) uint8
```

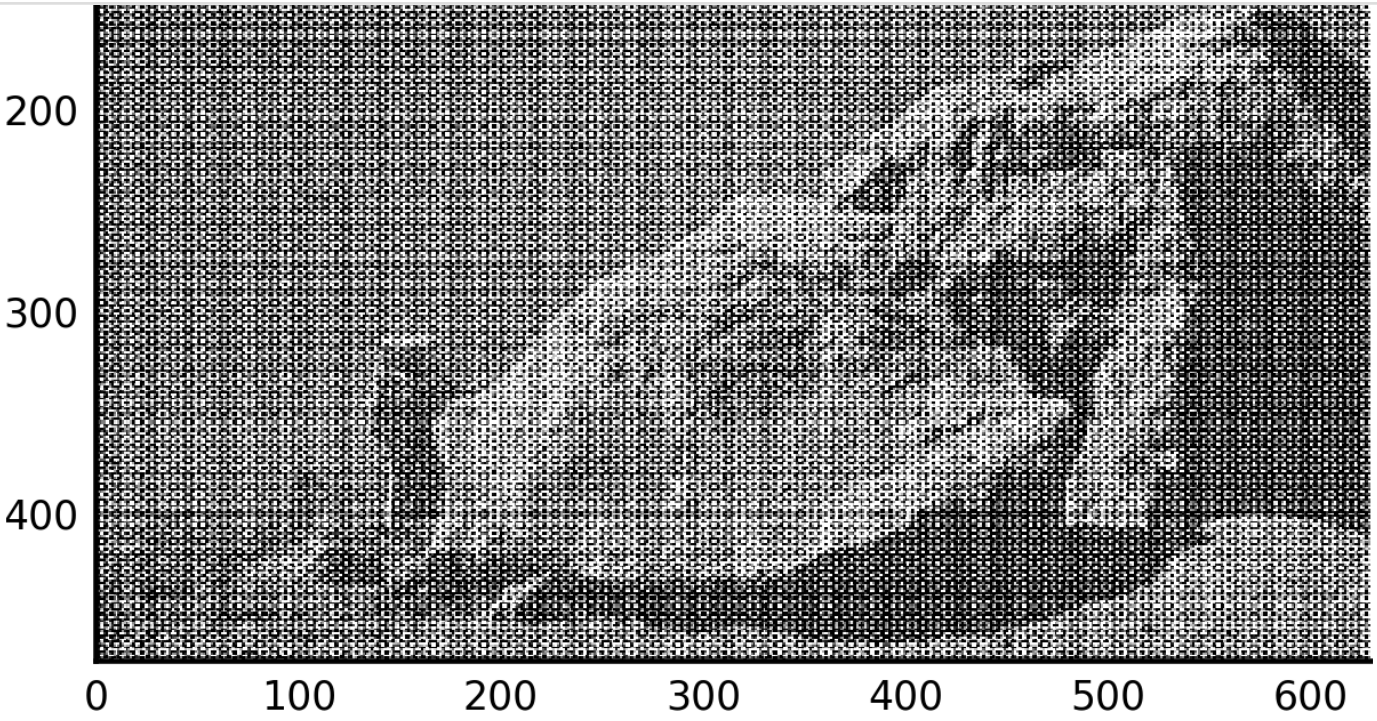


Figure 4-7. A noisy image of the moon landing

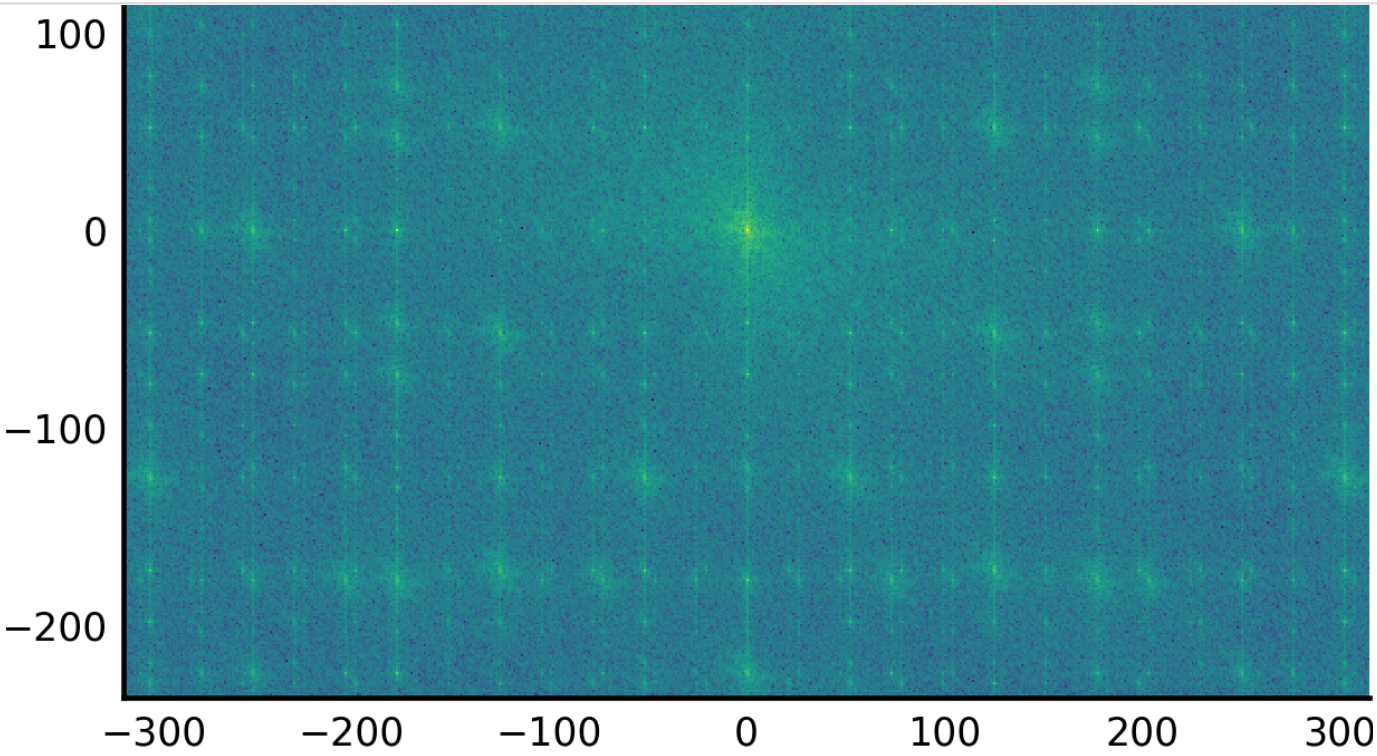
Do not adjust your monitor! The image you are seeing is real, although clearly distorted by either the measurement or transmission equipment.

To examine the spectrum of the image, we use `fftn` (instead of `fft`) to compute the DFT, since it has more than one dimension. The 2D FFT is equivalent to taking the 1D FFT across rows and then across columns, or vice versa.

```
F = fftpack.fftn(image)

F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)
```

Again, we take the log of the spectrum to compress the range of values, before displaying:

```
f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(np.log(1 + F_magnitude), cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```



Note the high values around the origin (middle) of the spectrum—these coefficients describe the low frequencies or smooth parts of the image, a vague canvas of the photo. Higher-frequency components, spread throughout the spectrum, fill in the edges and detail. Peaks around higher frequencies correspond to the periodic noise.

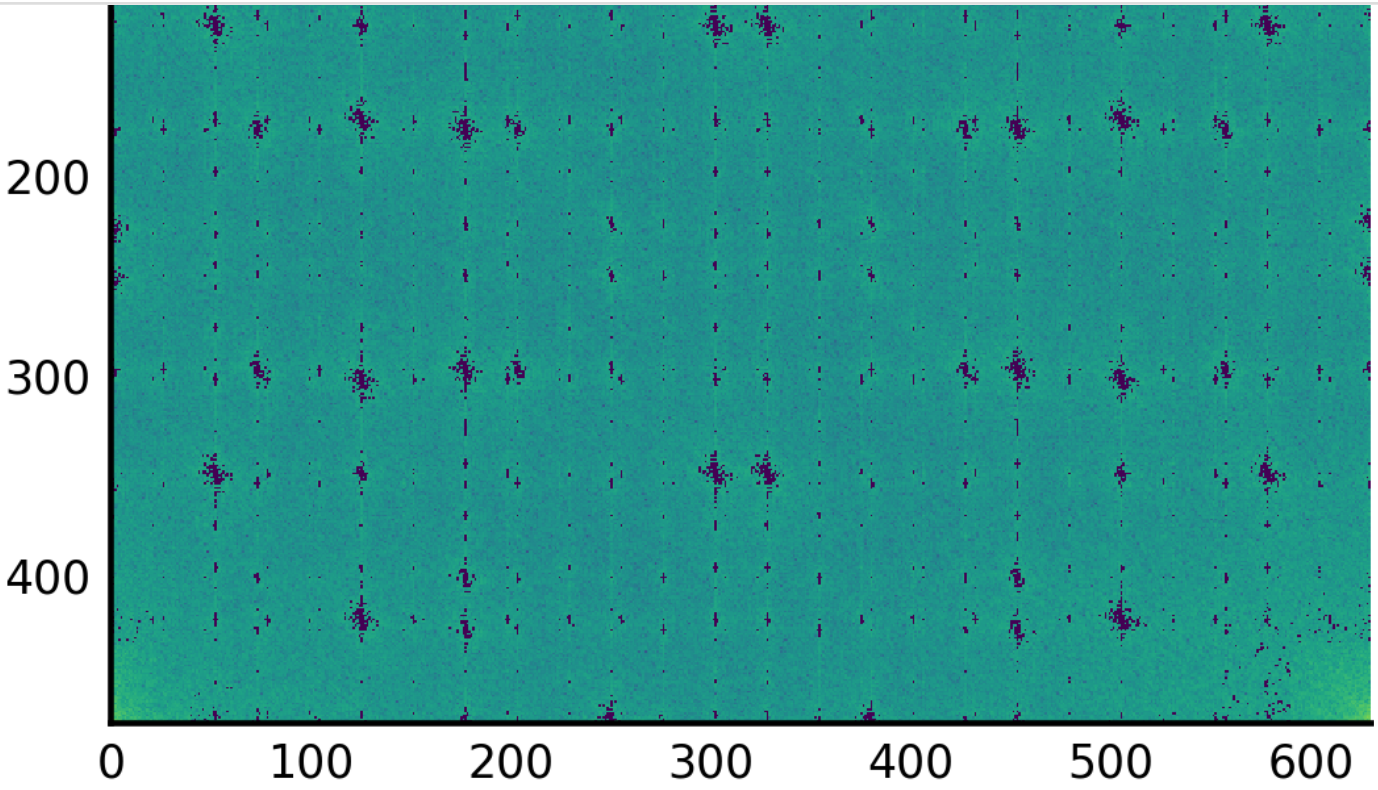
From the photo, we can see that the noise (measurement artifacts) is highly periodic, so we hope to remove it by zeroing out the corresponding parts of the spectrum (Figure 4-8).

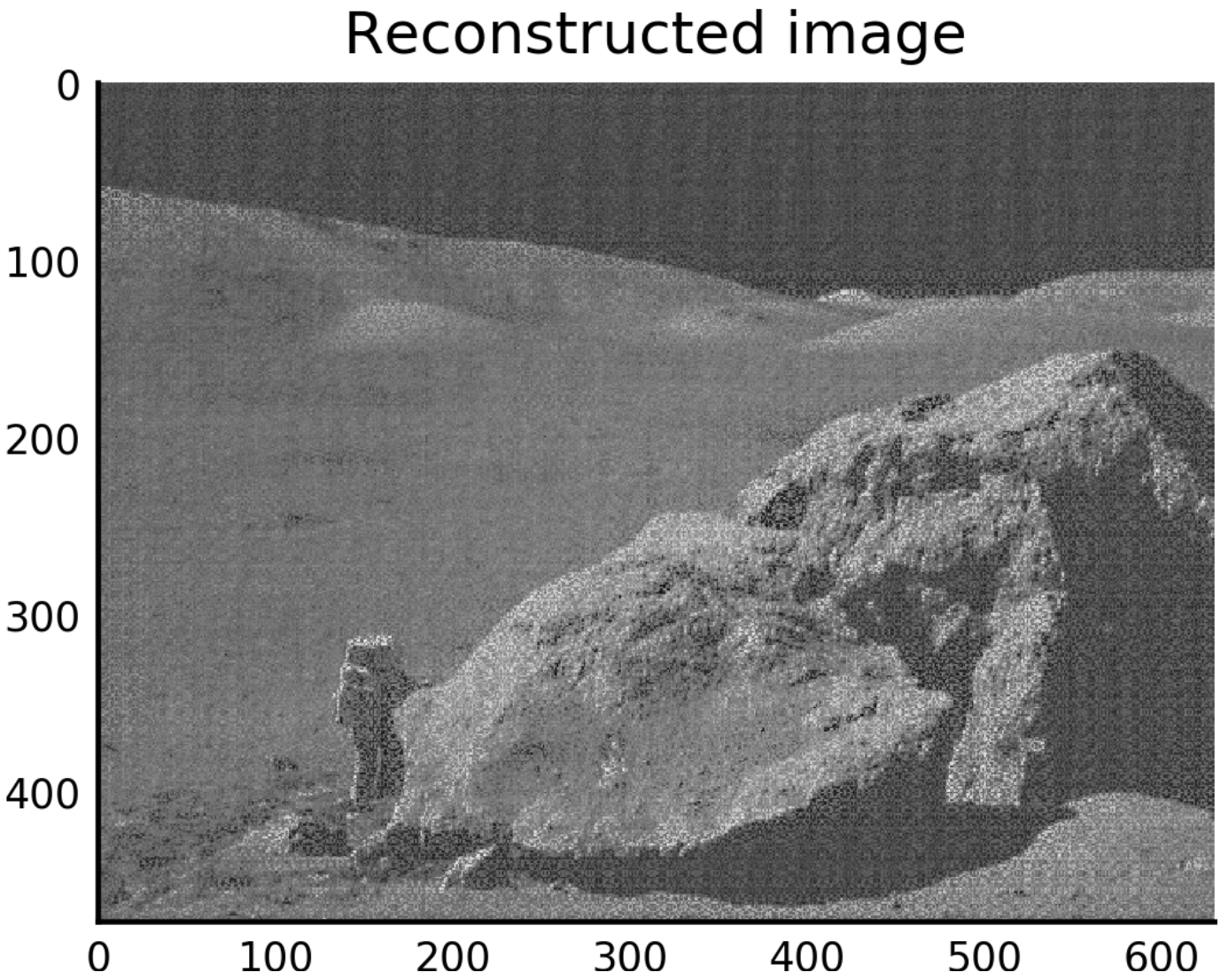
The image with those peaks suppressed indeed looks quite different!

```
# Set block around center of spectrum to zero
K = 40
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

# Find all peaks higher than the 98th percentile
peaks = F_magnitude < np.percentile(F_magnitude, 98)

# Shift the peaks back to align with the original spectrum
peaks = fftpack.ifftshift(peaks)

# Make a copy of the original (complex) spectrum
F_dim = F.copy()
```

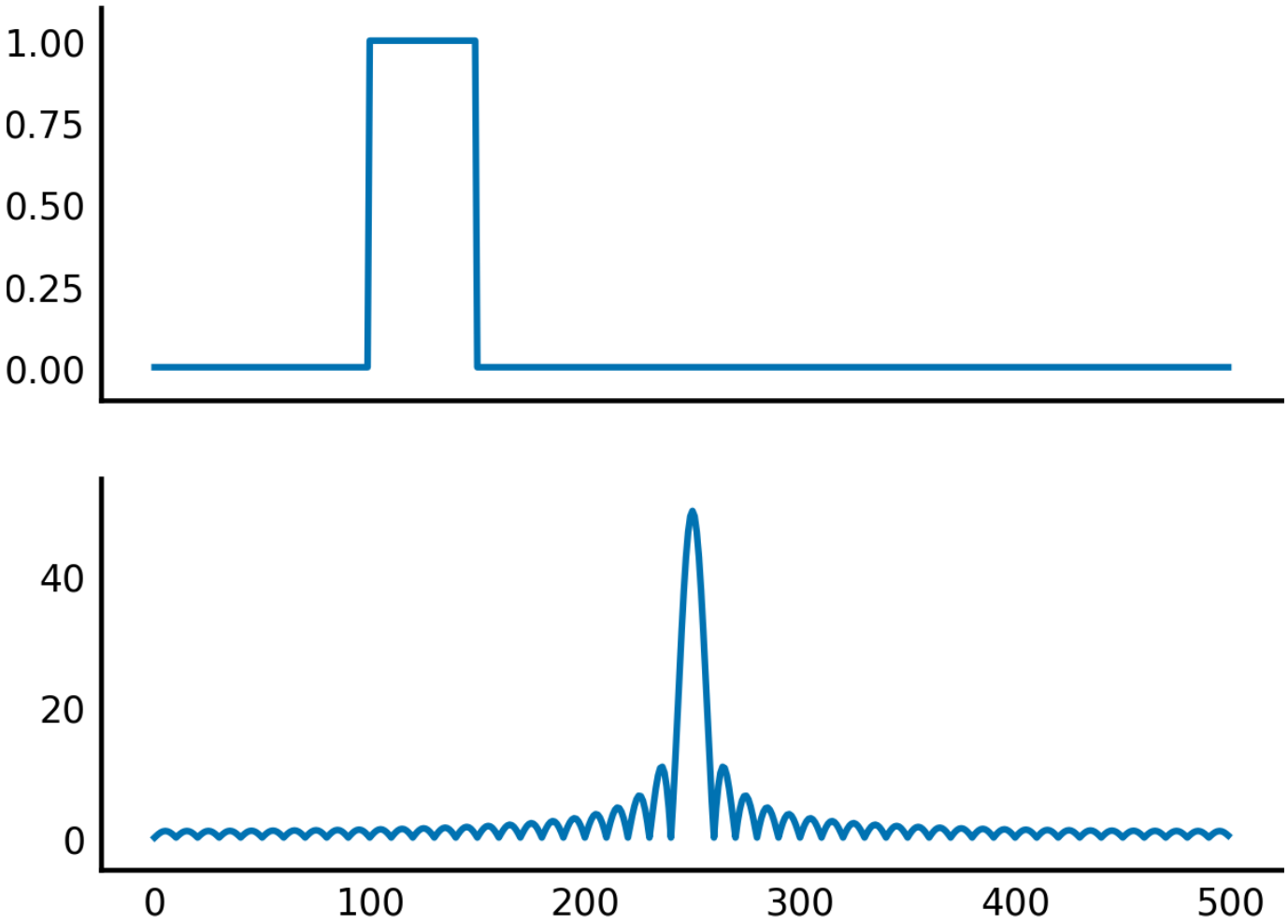
```
f, (ax0, ax1) = plt.subplots(2, 1, figsize=(4.8, 7))
ax0.imshow(np.log10(1 + np.abs(F_dim)), cmap='viridis')
ax0.set_title('Spectrum after suppression')

ax1.imshow(image_filtered)
ax1.set_title('Reconstructed image');
```

Elegant SciPy by Harriet Dashnow, Stéfan van der Walt, Juan Nunez-Iglesias



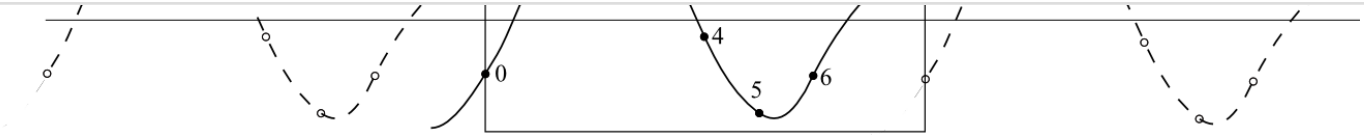


```
x = np.zeros(500)
x[100:150] = 1

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1, sharex=True)

ax0.plot(x)
ax0.set_ylim(-0.1, 1.1)

ax1.plot(fftpack.fftshift(np.abs(X)))
ax1.set_ylim(-5, 55);
```



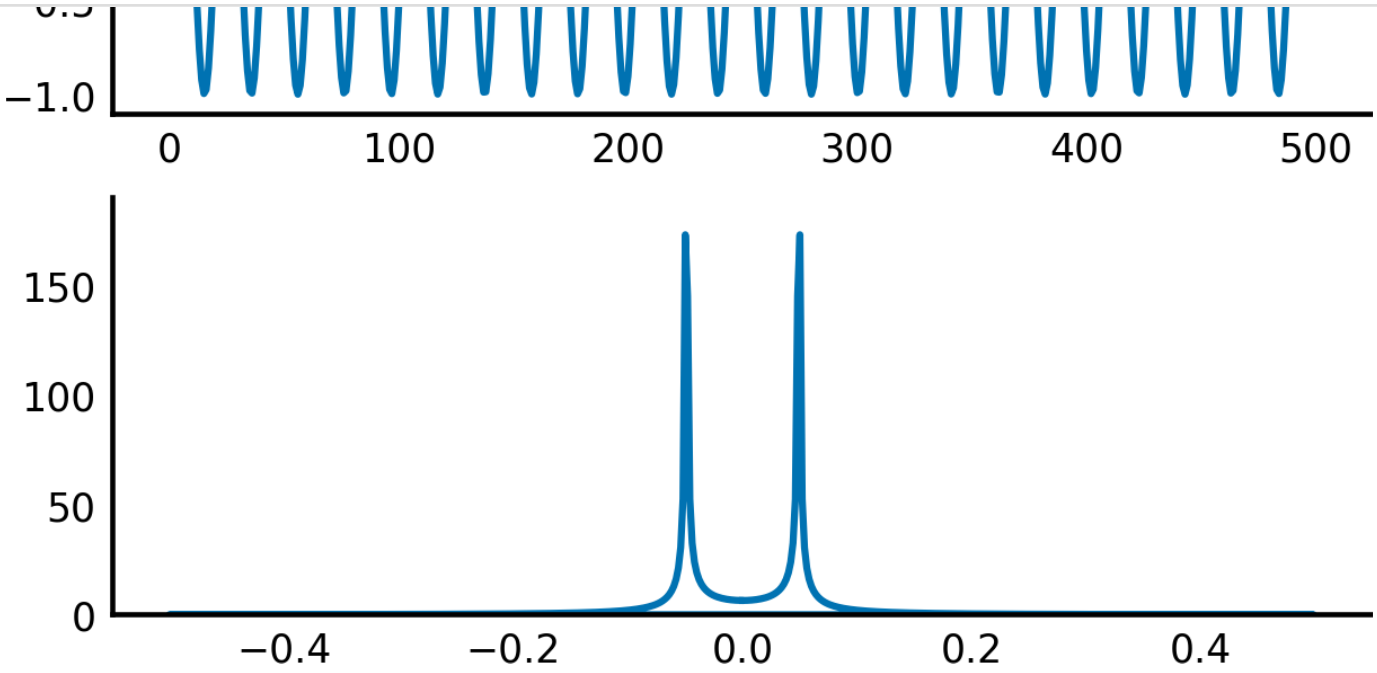
In theory, you would need a combination of infinitely many sinusoids (frequencies) to represent any abrupt transition; the coefficients would typically have the same side lobe structure as seen here for the pulse.

Importantly, the DFT assumes that the input signal is periodic. If the signal is not, the assumption is simply that, right at the end of the signal, it jumps back to its beginning value. Consider the function, *x(t)*,



We measure the signal for only a short time, labeled T_{eff}. The Fourier transform assumes that $x(8) = x(0)$, and that the signal is continued as the dashed, rather than the solid, line. This introduces a big jump at the edge, with the expected oscillation in the spectrum:

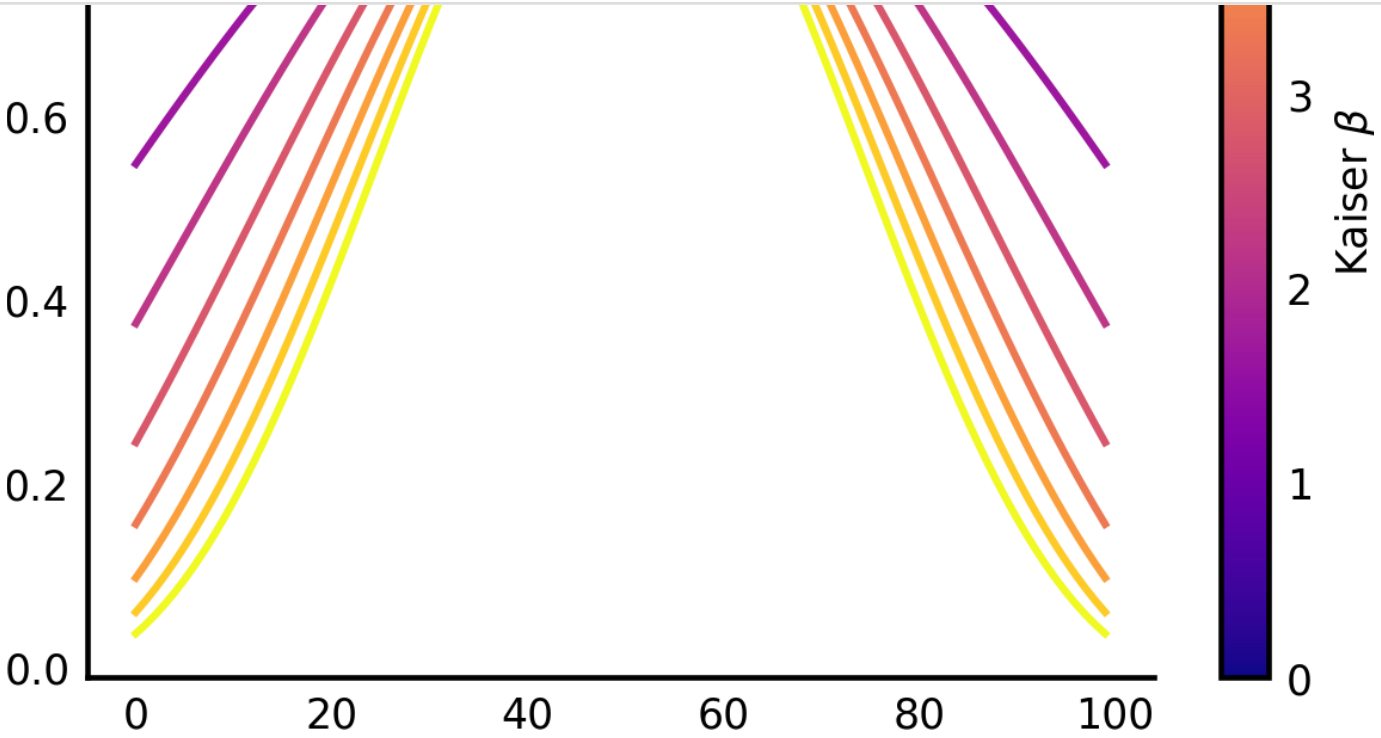
```
t = np.linspace(0, 1, 500)
x = np.sin(49 * np.pi * t)

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1)

ax0.plot(x)
ax0.set_ylim(-1.1, 1.1)

ax1.plot(fftpack.fftfreq(len(t)), np.abs(X))
ax1.set_ylim(0, 190);
```



Instead of the expected two lines, the peaks are spread out in the spectrum.

We can counter this effect by a process called *windowing*. The original function is multiplied with a window function such as the Kaiser window $K(N, \beta)$. Here we visualize it for β ranging from 0 to 100:

```
f, ax = plt.subplots()

N = 10
beta_max = 5
colormap = plt.cm.plasma

norm = plt.Normalize(vmin=0, vmax=beta_max)

lines = [
    ax.plot(np.kaiser(100, beta), color=colormap(norm(beta)))
    for beta in np.linspace(0, beta_max, N)
    ]

sm = plt.cm.ScalarMappable(cmap=colormap, norm=norm)

sm._A = []

plt.colorbar(sm).set_label(r'Kaiser $\beta$');
```

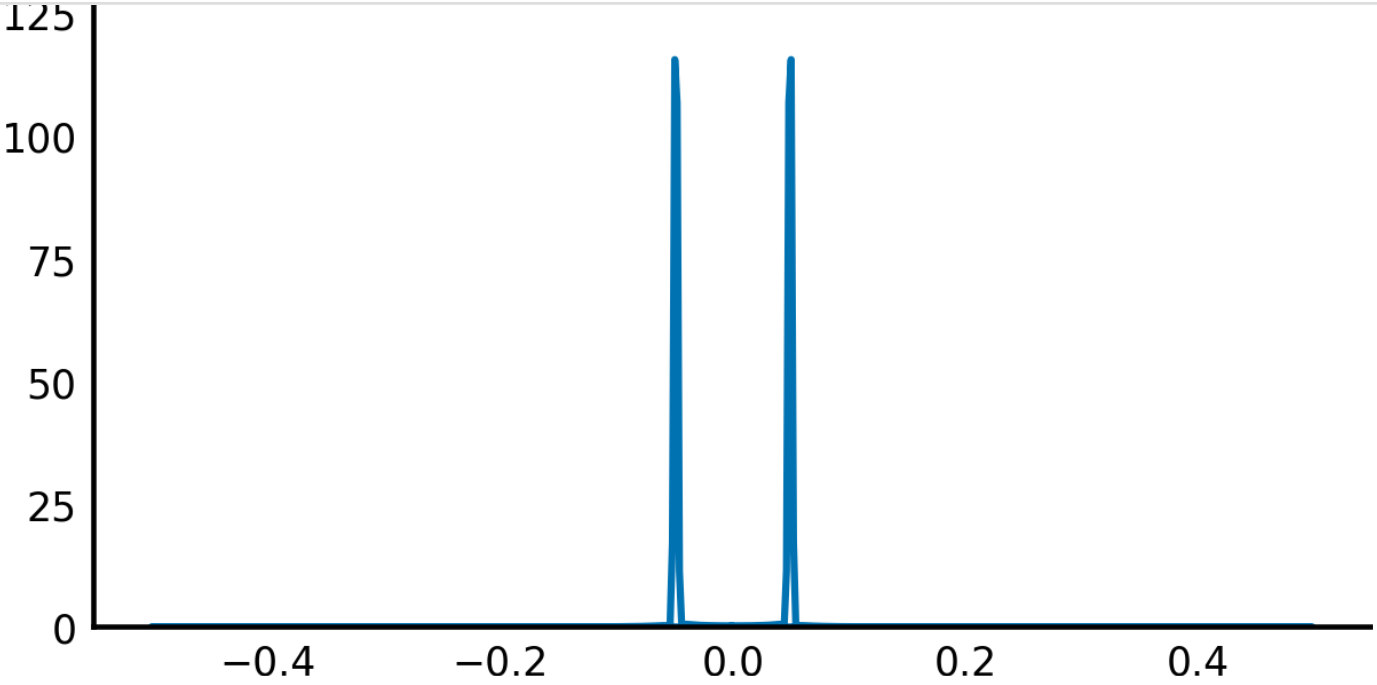


By changing the parameter β, we can change the shape of the window from rectangular ($\beta = 0$, no windowing) to a window that produces signals that smoothly increase from zero and decrease to zero at the endpoints of the sampled interval, producing very low side lobes (β typically between 5 and 10).[9]

Applying the Kaiser window here, we see that the side lobes have been drastically reduced, at the cost of a slight widening in the main lobe.

The effect of windowing our previous example is noticeable:

```
win = np.kaiser(len(t), 5)
X_win = fftpack.fft(x * win)

plt.plot(fftpack.fftfreq(len(t)), np.abs(X_win))
plt.ylim(0, 190);
```



Real-World Application: Analyzing Radar Data

Linearly modulated FMCW (Frequency-Modulated Continuous-Wave) radars make extensive use of the FFT algorithm for signal processing and provide examples of various applications of the FFT. We will use actual data from an FMCW radar to demonstrate one such application: target detection.

Roughly, an FMCW radar works like this (see "A Simple FMCW Radar System" and Figure 4-9 for more detail):

1. A signal with changing frequency is generated. This signal is transmitted by an antenna, after which it travels outward, away from the radar. When it hits an object, part of the signal is reflected back to the radar, where it is received, multiplied by a copy of the transmitted signal, and sampled, turning it into numbers that are packed into an array. Our challenge is to interpret those numbers to form meaningful results.

2. The preceding multiplication step is important. From school, recall the trigonometric identity:

$$\sin(xt)\sin(yt) = \frac{1}{2}\left[\sin\left((x-y)t + \frac{\pi}{2}\right) - \sin\left((x+y)t + \frac{\pi}{2}\right)\right]$$

3. Thus, if we multiply the received signal by the transmitted signal, we expect two frequency components to appear in the spectrum: one that is the difference in frequencies between the received and transmitted signal, and one that is the sum of their frequencies.

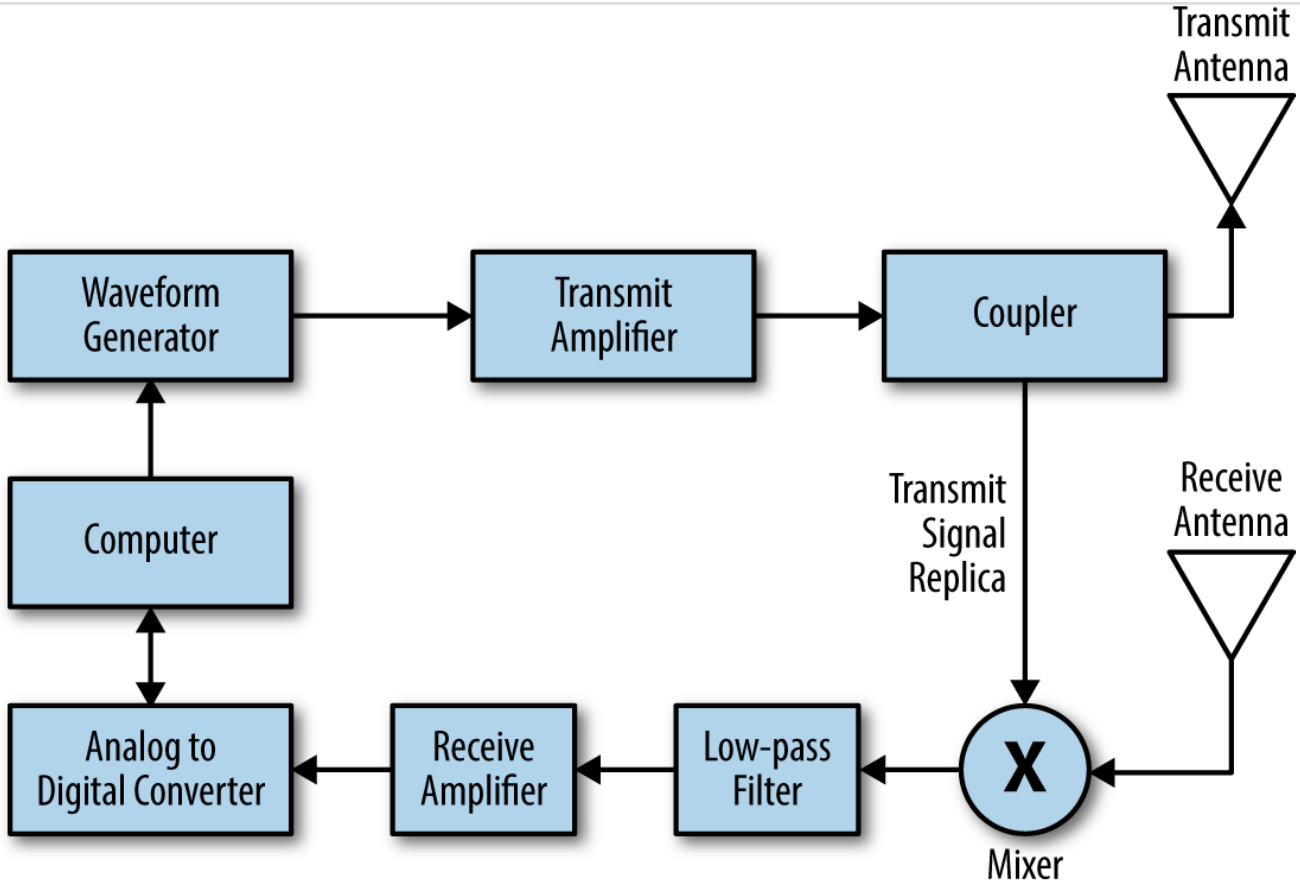


Figure 4-9. The block diagram of a simple FMCW radar system

A block diagram of a simple FMCW radar that uses separate transmit and receive antennas is shown above. The radar consists of a waveform generator that generates a sinusoidal signal of which the frequency varies linearly around the required transmit frequency. The generated signal is amplified to the required power level by the transmit amplifier and routed to the transmit antenna via a coupler circuit where a copy of the transmit signal is tapped off. The transmit antenna radiates the transmit signal as an electromagnetic wave in a narrow beam toward the target to be detected. When the wave encounters an object that reflects electromagnetic waves, a fraction of the energy irradiating the target is reflected back to the receiver as a second electromagnetic wave that propagates in the direction of the radar system. When this wave encounters the receive antenna, the antenna collects the energy in the wave energy impinging on it and converts it to a fluctuating voltage that is fed to the mixer. The mixer multiplies the received signal with a replica of the transmit signal and produces a sinusoidal signal with a frequency equal to the difference in frequency between the transmitted and received signals. The low-pass filter ensures that the received signal is band limited (i.e., does not contain frequencies that we don't care about) and the receive amplifier strengthens the signal to a suitable amplitude for the analog-to-digital converter (ADC) that feeds data to the computer.

To summarize, we should note that:

- The data that reaches the computer consists of N samples sampled (from the multiplied, filtered signal) at a sample frequency of f_s.
- The *amplitude* of the returned signal varies depending on the *strength of the reflection* (i.e., is a property of the target object and the distance between the target and the radar).

time, t, has passed, the frequency will now be tS higher (Figure 4-10). In that same time span, the radar signal has traveled $d = t/v$ meters, where v is the speed of the transmitted wave through air (roughly the same as the speed of light, 3×10^8 m/s).

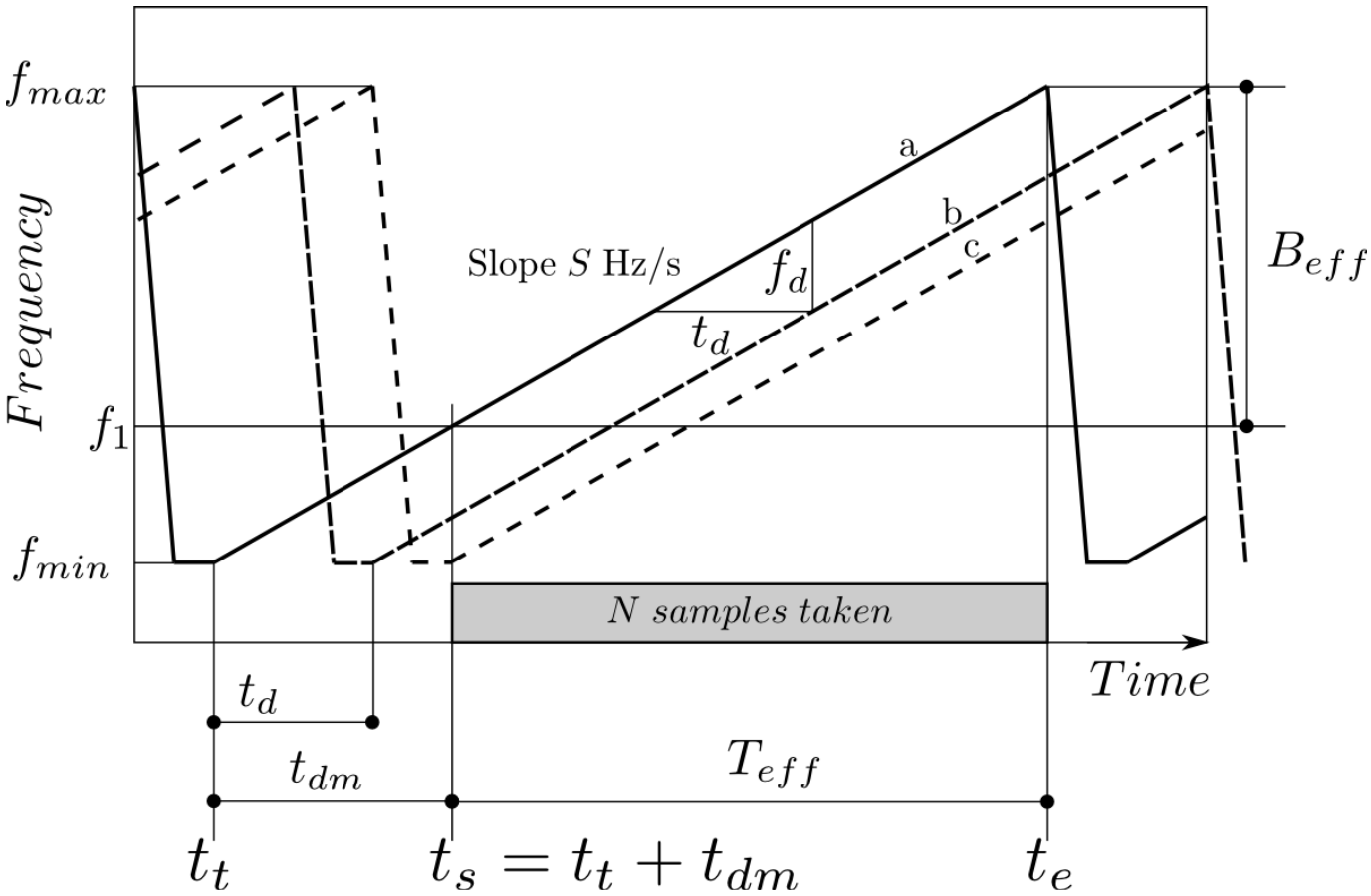


Figure 4-10. The frequency relationships in an FMCW radar with linear frequency modulation

Combining the above observations, we can calculate the amount of time it would take the signal to travel to, bounce off, and return from a target that is distance R away:

$t_R = 2R/v$

```
pi = np.pi

# Radar parameters
fs = 78125           # Sampling frequency in Hz, i.e., we sample 78125
                      # times per second

ts = 1 / fs          # Sampling time, i.e., one sample is taken each
                      # ts seconds

Teff = 2048.0 * ts  # Total sampling time for 2048 samples
```

```
# Specification of targets.  We made these targets up, imagining they
# are objects seen by the radar with the specified range and size.

R = np.array([100, 137, 154, 159,  180])  # Ranges (in meter)
M = np.array([0.33, 0.2, 0.9, 0.02, 0.1])  # Target size
P = np.array([0, pi / 2, pi / 3, pi / 5, pi / 6])  # Randomly chosen phase offsets

t = np.arange(2048) * ts  # Sample times

fd = 2 * S * R / 3E8      # Frequency differences for these targets

# Generate five targets
signals = np.cos(2 * pi * fd * t[:, np.newaxis] + P)

# Save the signal associated with the first target as an example for
# later inspection
v_single = signals[:, 0]

# Weigh the signals, according to target size and sum, to generate
# the combined signal seen by the radar.
v_sim = np.sum(M * signals, axis=1)

## The above code is equivalent to:
#
# v0 = np.cos(2 * pi * fd[0] * t)
# v1 = np.cos(2 * pi * fd[1] * t + pi / 2)
# v2 = np.cos(2 * pi * fd[2] * t + pi / 3)
# v3 = np.cos(2 * pi * fd[3] * t + pi / 5)
# v4 = np.cos(2 * pi * fd[4] * t + pi / 6)
#
## Blend them together
# v_single = v0
# v_sim = (0.33 * v0) + (0.2 * v1) + (0.9 * v2) + (0.02 * v3) + (0.1 * v4)
```

Here, we generated a synthetic signal, v_{single}, received when looking at a single target (see Figure 4-11). By counting the number of cycles seen in a given time period, we can compute the frequency of the signal and thus the distance to the target.

A real radar will rarely receive only a single echo, though. The simulated signal v_{sim} shows what a radar signal will look like with five targets at different ranges (including two close to one another at 154 and 159 meters), and $v_{\text{actual}}(t)$ shows the output signal obtained with an actual radar. When we add multiple echoes together, the result makes little intuitive sense (Figure 4-11); until, that is, we look at it more carefully through the lens of the DFT.

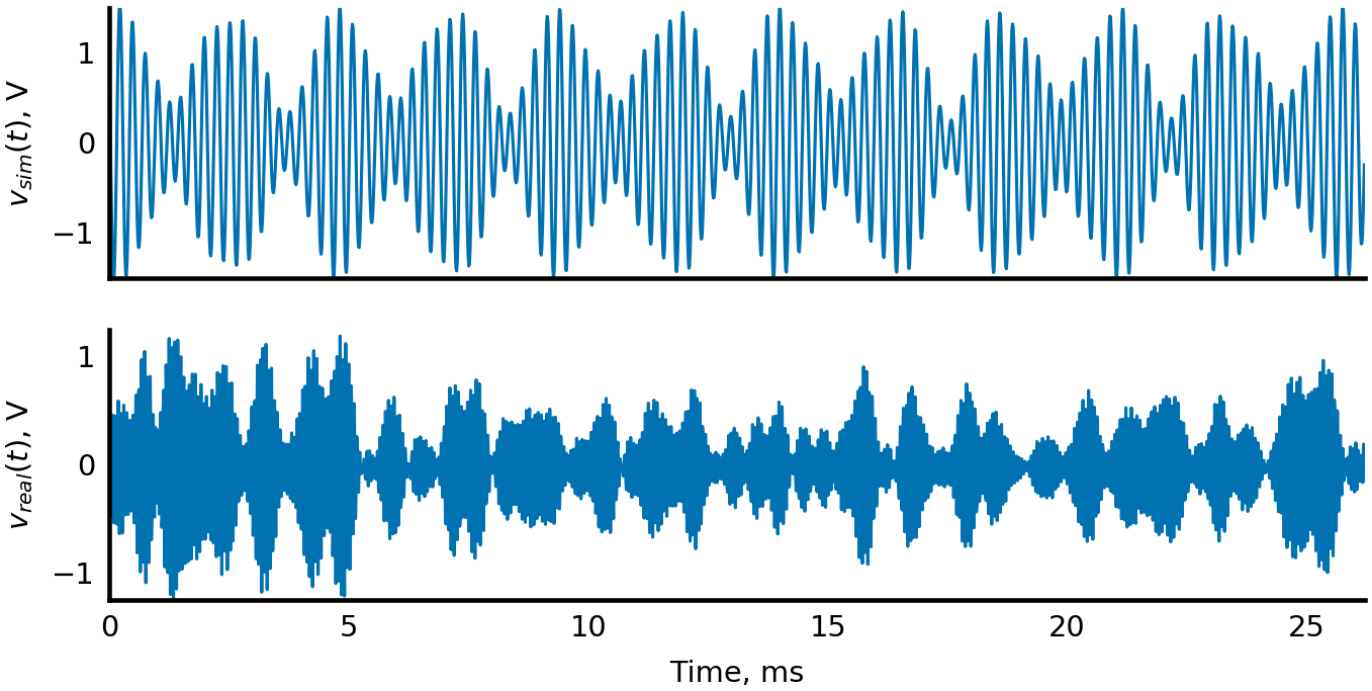


Figure 4-11. Receiver output signals: (a) single simulated target, (b) five simulated targets, and (c) actual radar data

The real-world radar data is read from a NumPy-format *.npz* file (a lightweight, cross-platform, and cross-version compatible storage format). These files can be saved with the `np.savez` or `np.savez_compressed` functions. Note that SciPy's `io` submodule can also easily read other formats, such as MATLAB and NetCDF files.

```
data = np.load('data/radar_scan_0.npz')

# Load variable 'scan' from 'radar_scan_0.npz'
scan = data['scan']

# The dataset contains multiple measurements, each taken with the
# radar pointing in a different direction.  Here we take one such as
# measurement, at a specified azimuth (left-right position) and elevation
# (up-down position).  The measurement has shape (2048,).

v_actual = scan['samples'][5, 14, :]

# The signal amplitude ranges from -2.5V to +2.5V.  The 14-bit
# analogue-to-digital converter in the radar gives out integers
# between -8192 to 8192.  We convert back to voltage by multiplying by
# $(2.5 / 8192)$.

v_actual = v_actual * (2.5 / 8192)
```

`size`

Unsigned 32-bit (4 byte) integer (`np.uint32`)

`position`

`az`

32-bit float (`np.float32`)

`el`

32-bit float (`np.float32`)

`region_type`

Unsigned 8-bit (1 byte) integer (`np.uint8`)

`region_ID`

Unsigned 16-bit (2 byte) integer (`np.uint16`)

`gain`

Unsigned 8-bit (1 byte) integer (`np.uin8`)

`samples`

2,048 unsigned 16-bit (2 byte) integers (`np.uint16`)

While it is true that NumPy arrays are *homogeneous* (i.e., all the elements inside are the same), it does not mean that those elements cannot be compound elements, as is the case here.

An individual field is accessed using dictionary syntax:

```
azimuths = scan['position']['az']  # Get all azimuth measurements
```

To summarize what we've seen so far: the shown measurements (v_{sim} and v_{actual}) are the sum of sinusoidal signals reflected by each of several objects. We need to determine each of the constituent components of

Elegant SciPy by Harriet Dashnow, Stéfan van der Walt, Juan Nunez-Iglesias

```
fig, axes = plt.subplots(3, 1, sharex=True, figsize=(4.8, 2.4))

# Take FFTs of our signals.  Note the convention to name FFTs with a
# capital letter.

V_single = np.fft.fft(v_single)
V_sim = np.fft.fft(v_sim)
V_actual = np.fft.fft(v_actual)

N = len(V_single)

with plt.style.context('style/thinner.mplstyle'):
    axes[0].plot(np.abs(V_single[:N // 2]))
    axes[0].set_ylabel("$|V_\mathrm{single}|$")
    axes[0].set_xlim(0, N // 2)
    axes[0].set_ylim(0, 1100)

    axes[1].plot(np.abs(V_sim[:N // 2]))
    axes[1].set_ylabel("$|V_\mathrm{sim} |$")
    axes[1].set_ylim(0, 1000)

    axes[2].plot(np.abs(V_actual[:N // 2]))
    axes[2].set_ylim(0, 750)
    axes[2].set_ylabel("$|V_\mathrm{actual}|$")

    axes[2].set_xlabel("FFT component $n$")

    for ax in axes:
        ax.grid()
```

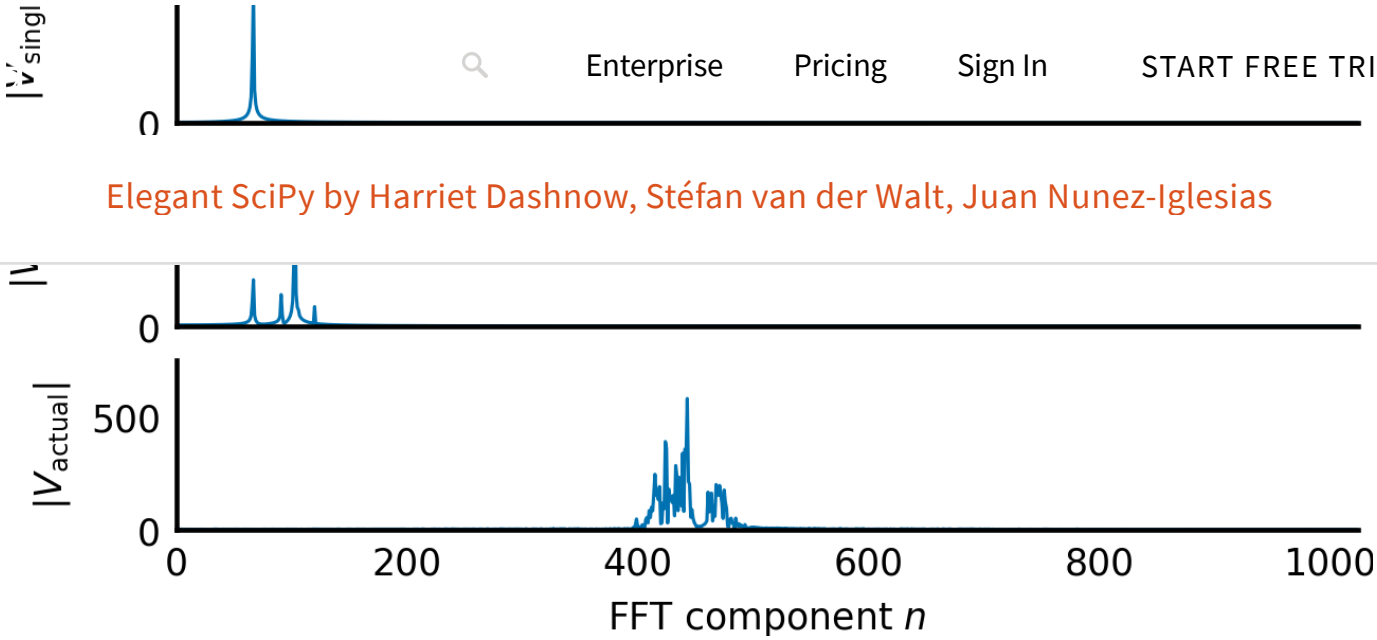


Figure 4-12. Range traces for: (a) single simulated target, (b) multiple simulated targets, and (c) real-world targets

Suddenly, the information makes sense!

The plot for $|V_0|$ clearly shows a target at component 67, and for $|V_{sim}|$ shows the targets that produced the signal that was uninterpretable in the time domain. The real radar signal, $|V_{actual}|$, shows a large number of targets between component 400 and 500 with a large peak in component 443. This happens to be an echo return from a radar illuminating the high wall of an open-cast mine.

To get useful information from the plot, we must determine the range! Again, we use the formula:

$$R_n = \frac{nv}{2B_{eff}}$$

In radar terminology, each DFT component is known as a *range bin*.

This equation also defines the range resolution of the radar: targets will only be distinguishable if they are separated by more than two range bins, for example:

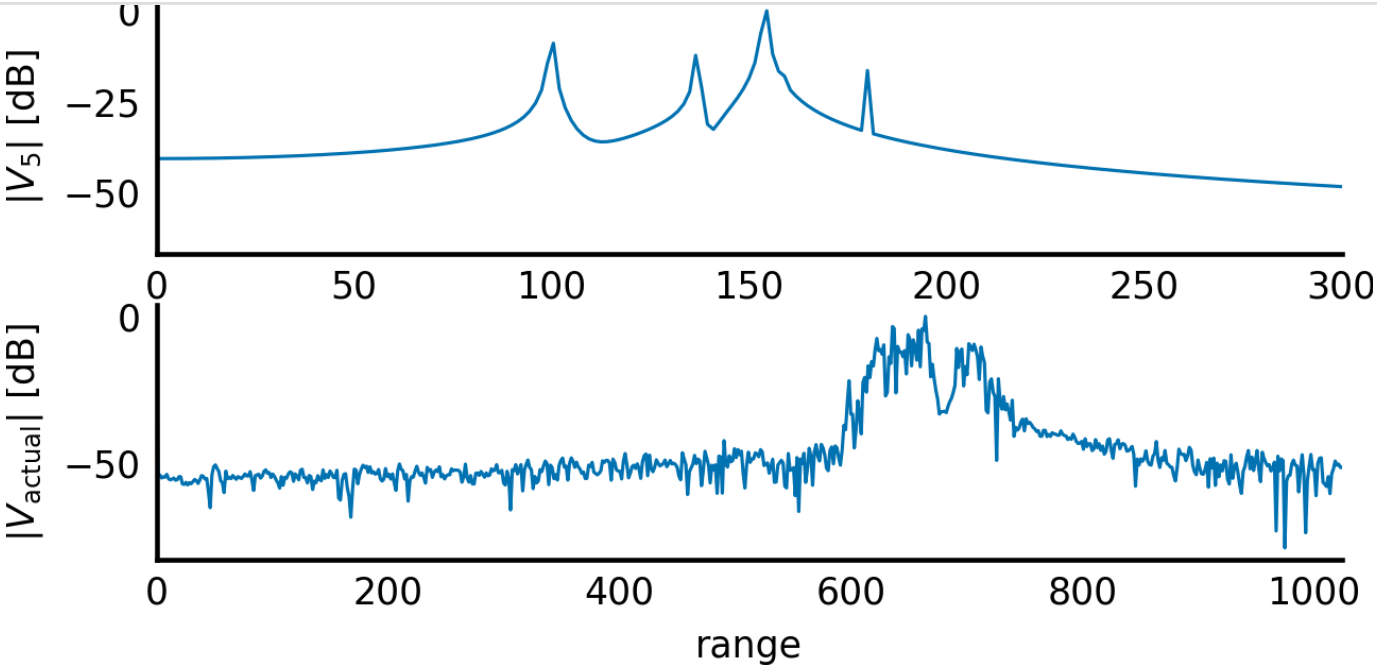
$$\Delta R > \frac{1}{B_{eff}}$$

This is a fundamental property of all types of radar.

This result is quite satisfying—but the dynamic range is so large that we could very easily miss some peaks. Let's take the log as before with the spectrogram:

```
c = 3e8  # Approximately the speed of light and of
```

```
    y = np.abs(y)
    y /= y.max()

    return 20 * np.log10(y)


def log_plot_normalized(x, y, ylabel, ax):
    ax.plot(x, dB(y))
    ax.set_ylabel(ylabel)
    ax.grid()


rng = np.arange(N // 2) * c / 2 / Beff

with plt.style.context('style/thinner.mplstyle'):
    log_plot_normalized(rng, V_single[:N // 2], "$|V_0|$ [dB]", ax0)
    log_plot_normalized(rng, V_sim[:N // 2], "$|V_5|$ [dB]", ax1)
    log_plot_normalized(rng, V_actual[:N // 2], "$|V_{\mathrm{actual}}|$ [dB]"
        , ax2)

ax0.set_xlim(0, 300)   # Change x limits for these plots so that
ax1.set_xlim(0, 300)   # we are better able to see the shape of the peaks.
ax2.set_xlim(0, len(V_actual) // 2)
ax2.set_xlabel('range')
```



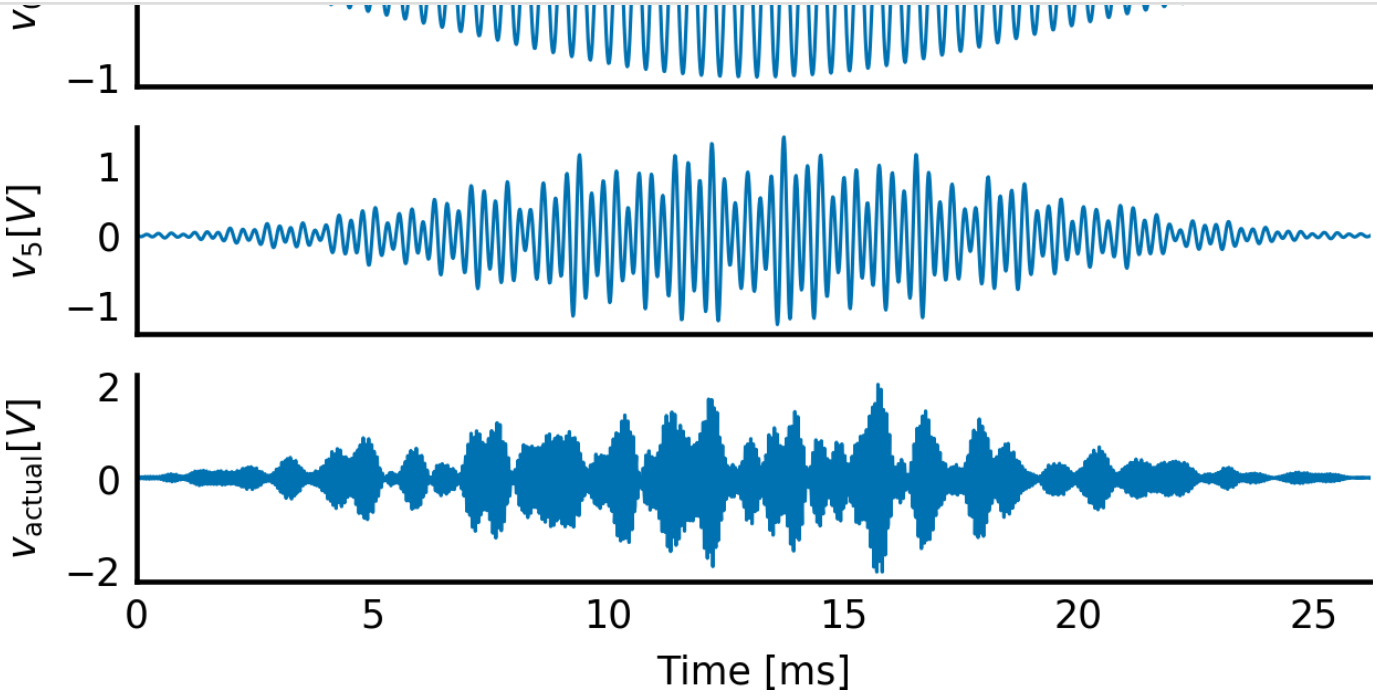
The observable dynamic range is much improved in these plots. For instance, in the real radar signal the *noise floor* of the radar has become visible (i.e., the level where electronic noise in the system starts to limit the radar's ability to detect a target).

Windowing, Applied

We're getting there, but in the spectrum of the simulated signal, we still cannot distinguish the peaks at 154 and 159 meters. Who knows what we're missing in the real-world signal! To sharpen the peaks, we'll return to our toolbox and make use of *windowing*.

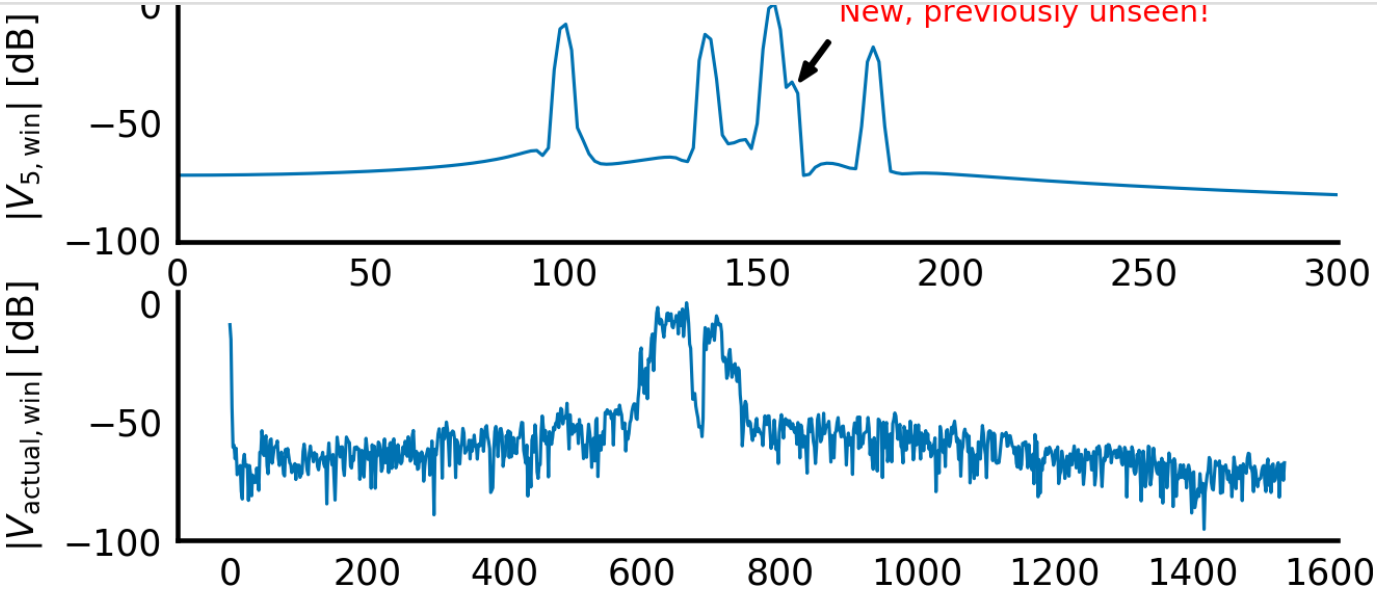
Here are the signals used thus far in this example, windowed with a Kaiser window with $\beta = 6.1$:

```
f, axes = plt.subplots(3, 1, sharex=True, figsize=(4.8, 2.8))

t_ms = t * 1000  # Sample times in milli-second

w = np.kaiser(N, 6.1)  # Kaiser window with beta = 6.1

for n, (signal, label) in enumerate([(v_single, r'$v_0 [V]$'),
                                     (v_sim, r'$v_5 [V]$'),
                                     (v_actual, r'$v_{\mathrm{actual}} [V]$')]):
    with plt.style.context('style/thinner.mplstyle'):
        axes[n].plot(t_ms, w * signal)
        axes[n].set_ylabel(label)
        axes[n].grid()
```



And the corresponding FFTs, or "range traces," in radar terms:

```
V_single_win = np.fft.fft(w * v_single)
V_sim_win = np.fft.fft(w * v_sim)
V_actual_win = np.fft.fft(w * v_actual)

fig, (ax0, ax1,ax2) = plt.subplots(3, 1)

with plt.style.context('style/thinner.mplstyle'):
    log_plot_normalized(rng, V_single_win[:N // 2],
                        r"$|V_{0,\mathrm{win}}|$ [dB]", ax0)
    log_plot_normalized(rng, V_sim_win[:N // 2],
                        r"$|V_{5,\mathrm{win}}|$ [dB]", ax1)
    log_plot_normalized(rng, V_actual_win[:N // 2],
                        r"$|V_\mathrm{actual,win}|$ [dB]", ax2)

ax0.set_xlim(0, 300)   # Change x limits for these plots so that
ax1.set_xlim(0, 300)   # we are better able to see the shape of the peaks.

ax1.annotate("New, previously unseen!", (160, -35), xytext=(10, 15),
             textcoords="offset points", color='red', size='x-small',
             arrowprops=dict(width=0.5, headwidth=3, headlength=4,
                             fc='k', shrink=0.1));
```



Compare these with the earlier range traces. There is a dramatic lowering in side-lobe level, but at a price: the peaks have changed in shape, widening and becoming less peaky, thus lowering the radar resolution—that is, the ability of the radar to distinguish between two closely spaced targets. The choice of window is a compromise between side-lobe level and resolution. Even so, referring to the trace for V_{sim}, windowing has dramatically increased our ability to distinguish the small target from its large neighbor.

In the real radar data range trace, windowing has also reduced the side lobes. This is most visible in the depth of the notch between the two groups of targets.

Radar Images

Knowing how to analyze a single trace, we can expand to looking at radar images.

The data is produced by a radar with a parabolic reflector antenna. It produces a highly directive round pencil beam with a two-degree spreading angle between half-power points. When directed with normal incidence at a plane, the radar will illuminate a spot of about 2m in diameter at a distance of 60m. Outside this spot, the power drops off quite rapidly, but strong echoes from outside the spot will nevertheless still be visible.

By varying the pencil beam's azimuth (left-right position) and elevation (up-down position), we can sweep it across the target area of interest. When reflections are picked up, we can calculate the distance to the reflector (the object hit by the radar signal). Together with the current pencil beam azimuth and elevation, this defines the reflector's position in 3D.

A rock slope consists of thousands of reflectors. A range bin can be thought of as a large sphere with the radar at its center that intersects the slope along a ragged line. The scatterers on this line will produce re-

degrees.

We will now draw some contour plots of the resulting radar data. Refer to Figure 4-13 to see how the different slices are taken. A first slice at fixed range shows the strength of echoes against elevation and azimuth. Another two slices at fixed elevation and azimuth, respectively, show the slope (see Figures 4-13 and 4-14). The stepped construction of the high wall in an opencast mine is visible in the azimuth plane.

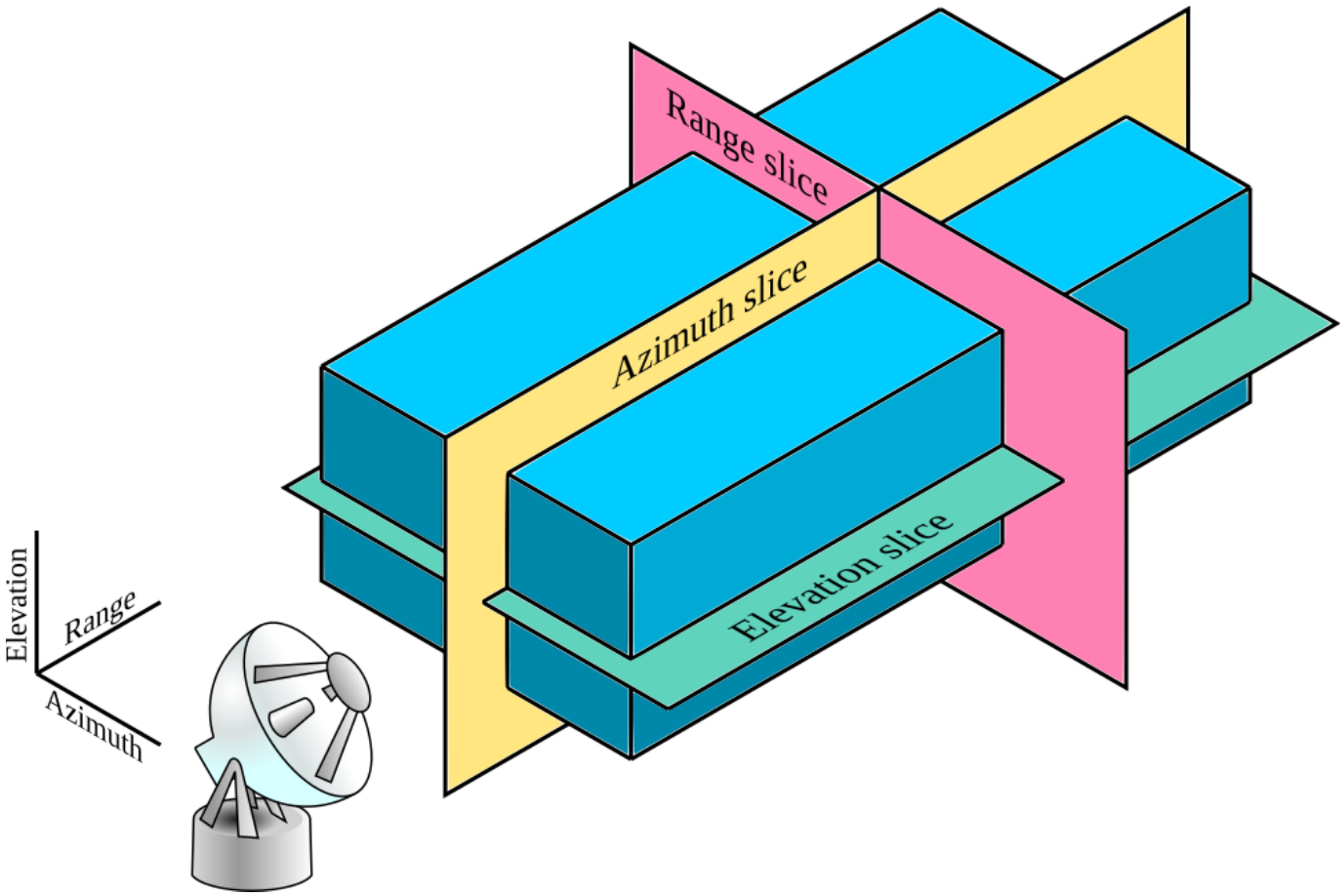


Figure 4-13. Diagram showing azimuth, elevation, and range slices through data volume

```
data = np.load('data/radar_scan_1.npz')
scan = data['scan']

# The signal amplitude ranges from -2.5V to +2.5V.  The 14-bit
# analogue-to-digital converter in the radar gives out integers
# between -8192 to 8192.  We convert back to voltage by multiplying by
# $(2.5 / 8192)$.

v = scan['samples'] * 2.5 / 8192
win = np.hanning(N + 1)[:-1]

# Take FFT for each measurement
```

Elegant SciPy by Harriet Dashnow, Stéfan van der Walt, Juan Nunez-Iglesias

```
f, axes = plt.subplots(2, 2, figsize=(4.8, 4.8), tight_layout=True)

labels = ('Range', 'Azimuth', 'Elevation')

def plot_slice(ax, radar_slice, title, xlabel, ylabel):
    ax.contourf(dB(radar_slice), contours, cmap='magma_r')
    ax.set_title(title)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_facecolor(plt.cm.magma_r(-40))

with plt.style.context('style/thinner.mplstyle'):
    plot_slice(axes[0, 0], V[:, :, 250], 'Range=250', 'Azimuth', 'Elevation')
    plot_slice(axes[0, 1], V[:, 3, :], 'Azimuth=3', 'Range', 'Elevation')
    plot_slice(axes[1, 0], V[6, :, :].T, 'Elevation=6', 'Azimuth', 'Range')
    axes[1, 1].axis('off')
```

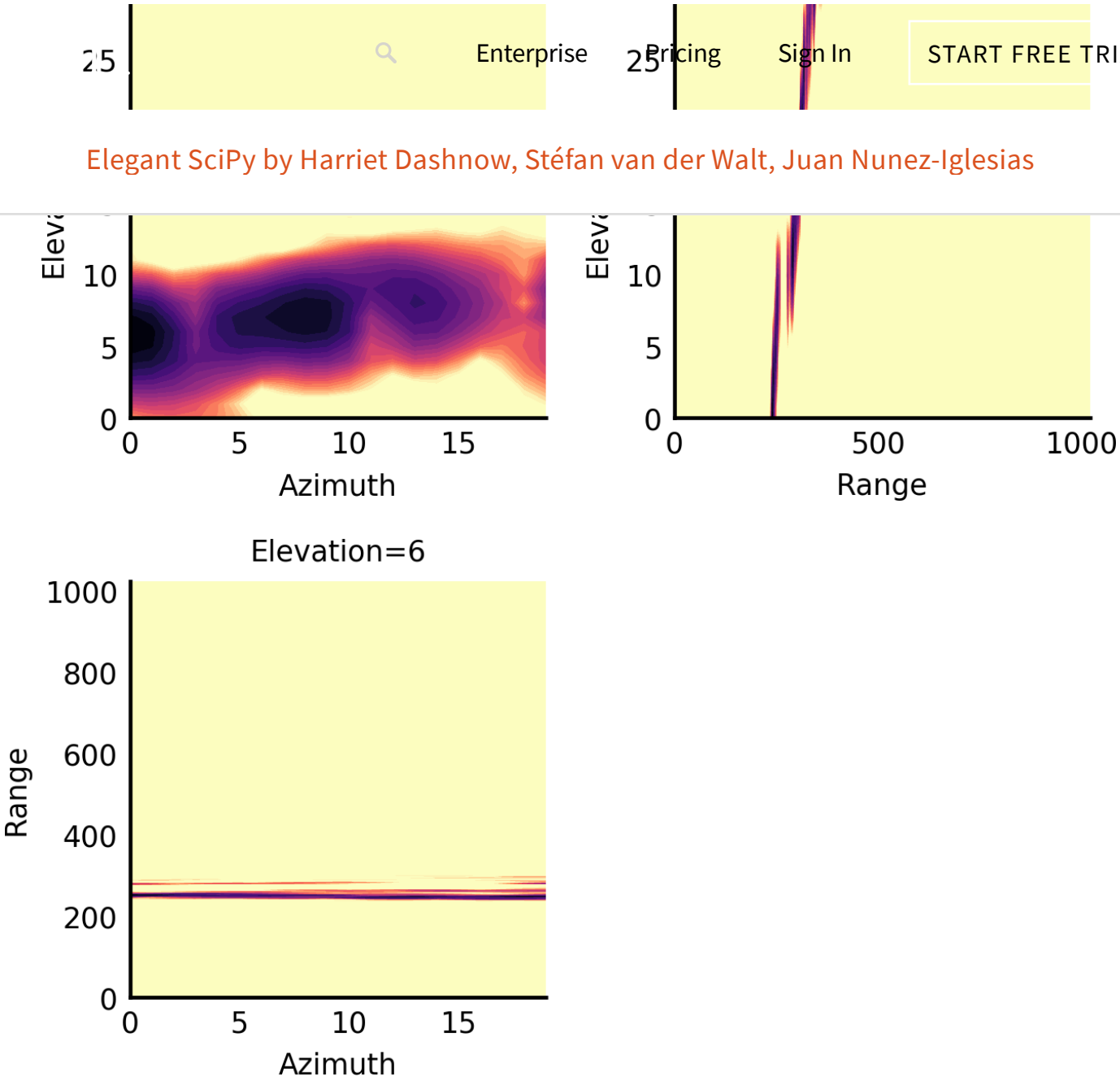


Figure 4-14. Contour plots of range traces along various axes (see Figure 4-13)

3D visualization

We can also visualize the volume in three dimensions (Figure 4-15).

We first compute the argmax (the index of the maximum value) in the range direction. This should give an indication of the range at which the radar beam hit the rock slope. Each argmax index is converted to a 3D (elevation-azimuth-range) coordinate:

```
r = np.argmax(V, axis=2)

el, az = np.meshgrid(*[np.arange(s) for s in r.shape], indexing='ij')

axis_labels = ['Elevation', 'Azimuth', 'Range']
coords = np.column_stack((el.flat, az.flat, r.flat))
```

```
d = spatial.Delaunay(coords[:, :2])
simplexes = coords[d.vertices]
```

For display purposes, we swap the range axis to be the first:

```
coords = np.roll(coords, shift=-1, axis=1)
axis_labels = np.roll(axis_labels, shift=-1)
```

Now, Matplotlib's `trisurf` can be used to visualize the result:

```
# This import initializes Matplotlib's 3D machinery
from mpl_toolkits.mplot3d import Axes3D

# Set up the 3D axis
f, ax = plt.subplots(1, 1, figsize=(4.8, 4.8),
                         subplot_kw=dict(projection='3d'))

with plt.style.context('style/thinner.mplstyle'):
    ax.plot_trisurf(*coords.T, triangles=d.vertices, cmap='magma_r')

    ax.set_xlabel(axis_labels[0])
    ax.set_ylabel(axis_labels[1])
    ax.set_zlabel(axis_labels[2], labelpad=-3)
    ax.set_xticks([0, 5, 10, 15])

# Adjust the camera position to match our diagram above
ax.view_init(azim=-50);
```

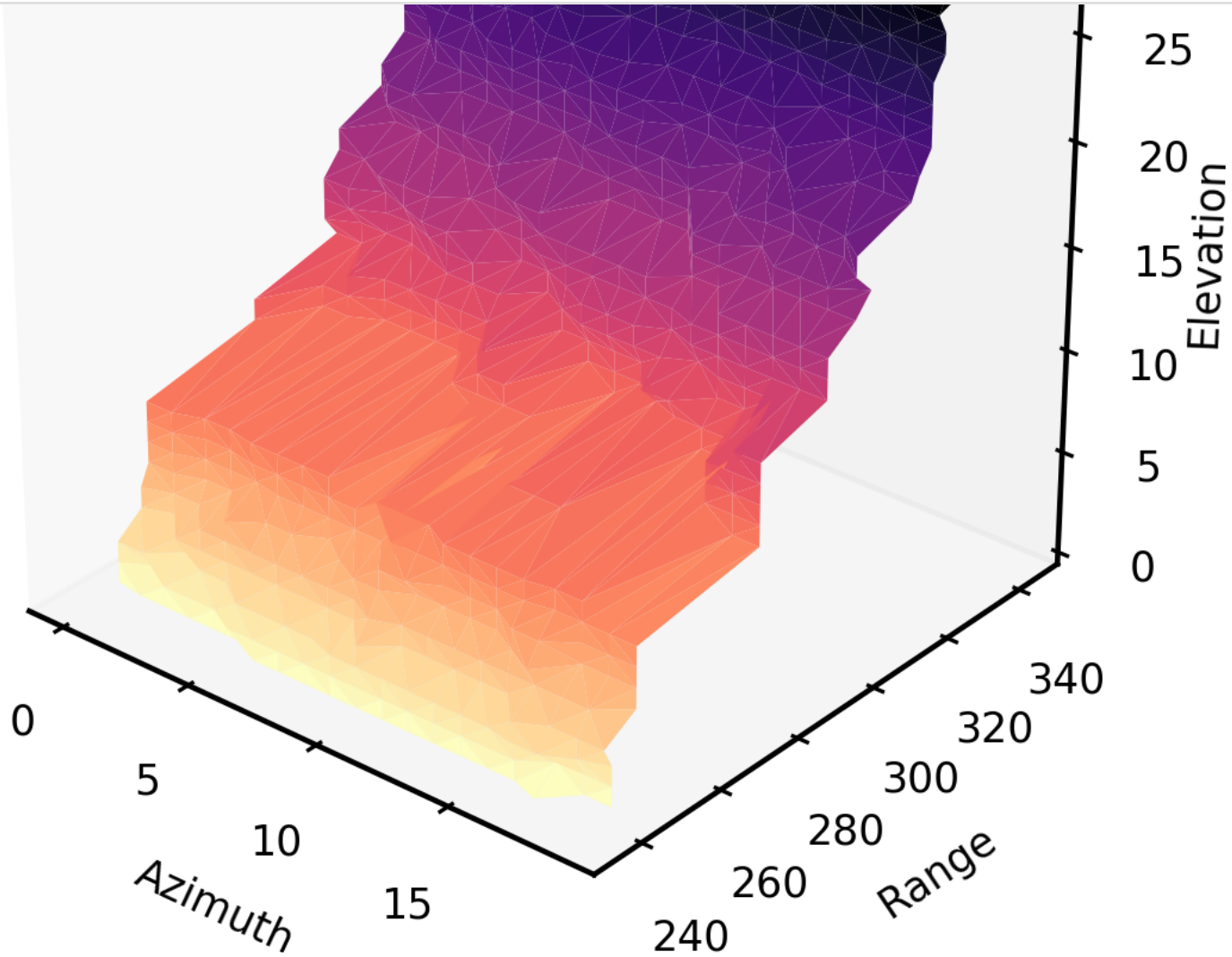


Figure 4-15. 3D visualization of estimated rock slope position

Further Applications of the FFT

The preceding examples show just one of the uses of the FFT in radar. There are many others, such as movement (Doppler) measurement and target recognition. The FFT is pervasive, and is seen everywhere from MRI to statistics. With the basic techniques that this chapter outlines in hand, you should be well equipped to use it!

Further Reading

On the Fourier transform:

- Athanasios Papoulis, *The Fourier Integral and Its Applications* (New York: McGraw-Hill, 1960).
- Ronald A. Bracewell, *The Fourier Transform and Its Applications* (New York: McGraw-Hill, 1986).

Exercise: Image Convolution

The FFT is often used to speed up image convolution (convolution is the application of a sliding filter). Convolve an image with `np.ones((5, 5))`, using a) NumPy's `np.convolve` and b) `np.fft.fft2`. Confirm that the results are identical.

Hints:

- The convolution of `x` and `y` is equivalent to `ifft2(X * Y)`, where `X` and `Y` are the FFTs of x and y, respectively.
- In order to multiply `X` and `Y`, they have to be the same size. Use `np.pad` to extend `x` and `y` with zeros (toward the right and bottom) *before* taking their FFT.
- You may see some edge effects. You can remove these by increasing the padding size, so that both `x` and `y` have dimensions `shape(x) + shape(y) - 1`.

Check out "Solution: Image Convolution".

---

[1] The DFT operates on sampled data, in contrast to the standard Fourier transform, which is defined for continuous functions.

[2] The Fourier transform essentially tells us how to combine a set of sinusoids of varying frequency to form the input signal. The spectrum consists of complex numbers—one for each sinusoid. A complex number encodes two things: a magnitude and an angle. The magnitude is the strength of the sinusoid in the signal, and the angle is how much it is shifted in time. At this point, we only care about the magnitude, which we calculate using `np.abs`.

[3] For more on techniques for calculating both (approximate) frequencies and time of occurrence, read up on wavelet analysis.

[4] SciPy goes to some effort to preserve the energy in the spectrum. Therefore, when taking only half the components (for N even), it multiplies the remaining components, apart from the first and last components, by two (those two components are "shared" by the two halves of the spectrum). It also normalizes the window by dividing it by its sum.

[5] The period can, in fact, also be infinite! The general continuous Fourier transform provides for this possibility. DFTs are generally defined over a finite interval, and this is implicitly the period of the time domain function that is transformed. In other words, if you do the inverse DFT, you *always* get a periodic signal out.

computational cost grows quadratically with the number of input elements.

[7] While ideally we don't want to reimplement existing algorithms, sometimes it becomes necessary in order to obtain the best execution speeds possible, and tools like Cython, which compiles Python to C, and Numba, which does just-in-time compilation of Python code, make life a lot easier (and faster!). If you are able to use GPL-licensed software, you may consider using PyFFTW for faster FFTs.

[8] We leave it as an exercise for the reader to picture the situation for N odd. In this chapter, all examples use even-order DFTs.

[9] The classical windowing functions include Hann, Hamming, and Blackman. They differ in their side lobe levels and in the broadening of the main lobe (in the Fourier domain). A modern and flexible window function that is close to optimal for most applications is the Kaiser window—a good approximation to the optimal prolate spheroid window, which concentrates the most energy into the main lobe. We can tune the Kaiser window to suit the particular application, as illustrated in the main text, by adjusting the parameter β.