Practical Introduction to Time-Frequency Analysis

Open Script

This example shows how to perform and interpret basic time-frequency signal analysis. In practical applications, many signals are nonstationary. This means that their frequency-domain representation (their spectrum) changes over time. The example discusses the advantages of using time-frequency techniques over frequency-domain or time-domain representations of a signal. It answers basic questions, such as: When is a particular frequency component present in my signal? How do I increase time or frequency resolution? How can I sharpen the spectrum of a component or extract a particular mode? How do I measure power in a time-frequency representation? How do I visualize the time-frequency information of my signal?

Using Time-Frequency Analysis to Identify Numbers in a DTMF Signal

You can divide almost any time-varying signal into time intervals short enough that the signal is essentially stationary in each section. Time-frequency analysis is most commonly performed by segmenting a signal into those short periods and estimating the spectrum over sliding windows. The `spectrogram` function computes an FFT-based spectral estimate over each sliding window and lets you visualize how the frequency content of the signal changes over time.

Consider the signaling system of a digital phone dial. The signals produced by such a system are known as dual-tone multi-frequency (DTMF) signals. The particular sound generated by each dialed number consists of the sum of two sinusoids -- or tones -- with frequencies taken from two mutually exclusive groups. Each pair of tones contains one frequency of the low group (697 Hz, 770 Hz, 852 Hz, or 941 Hz) and one frequency of the high group (1209 Hz, 1336 Hz, or 1477Hz) and represents a unique symbol. The following are the frequencies allocated to the buttons of a telephone pad:

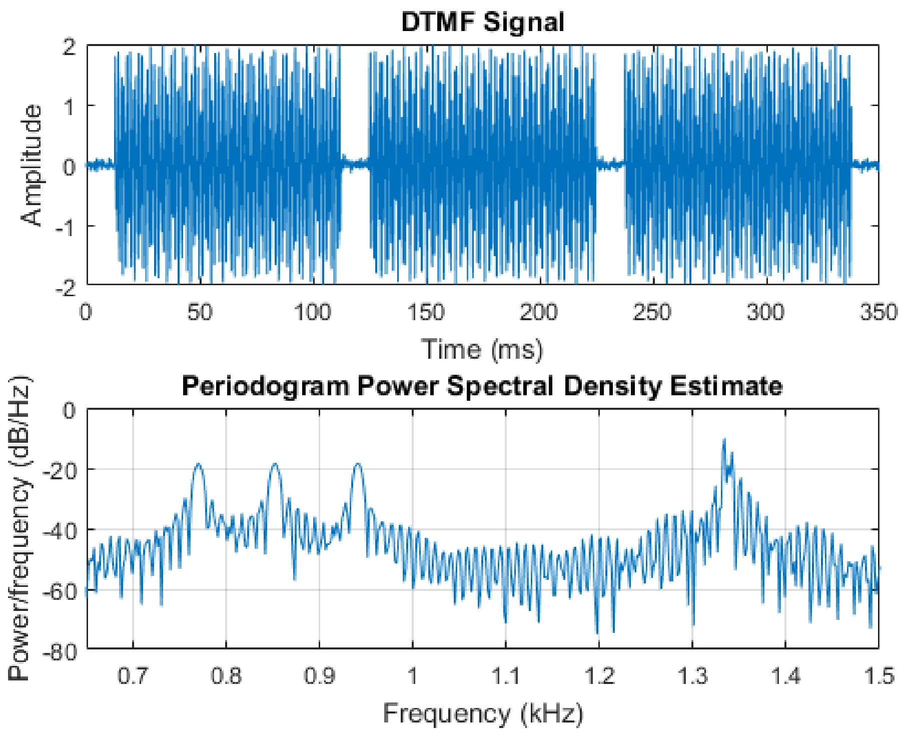
```
          1209 Hz   1336 Hz   1477 Hz
         _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
        |         |         |         |
        |         |   ABC   |   DEF   |
697 Hz  |    1    |    2    |    3    |
        |_ _ _ _ _|_ _ _ _ _|_ _ _ _ _|
        |         |         |         |
        |   GHI   |   JKL   |   MNO   |
770 Hz  |    4    |    5    |    6    |
        |_ _ _ _ _|_ _ _ _ _|_ _ _ _ _|
        |         |         |         |
        |   PRS   |   TUV   |   WXY   |
852 Hz  |    7    |    8    |    9    |
        |_ _ _ _ _|_ _ _ _ _|_ _ _ _ _|
        |         |         |         |
        |         |         |         |
941 Hz  |    *    |    0    |    #    |
        |_ _ _ _ _|_ _ _ _ _|_ _ _ _ _|
```

Generate a DTMF signal and listen to it.

```
[tones, Fs] = helperDTMFToneGenerator();
audid = audiodevinfo(0,Fs,16,1);
if  audid ~= -1
    p = audioplayer(tones,Fs,16,audid);
    play(p)
end
```

Listening to the signal, you can tell that a three-digit number was dialed. However, you cannot tell which number it was. Next, visualize the signal in both time and frequency domains.

```
N = numel(tones);
t = (0:N-1)/Fs;
subplot(2,1,1)
plot(1e3*t,tones)
xlabel('Time (ms)')
ylabel('Amplitude')
title('DTMF Signal')
subplot(2,1,2)
periodogram(tones,[],[],Fs)
xlim([0.65 1.5])
```

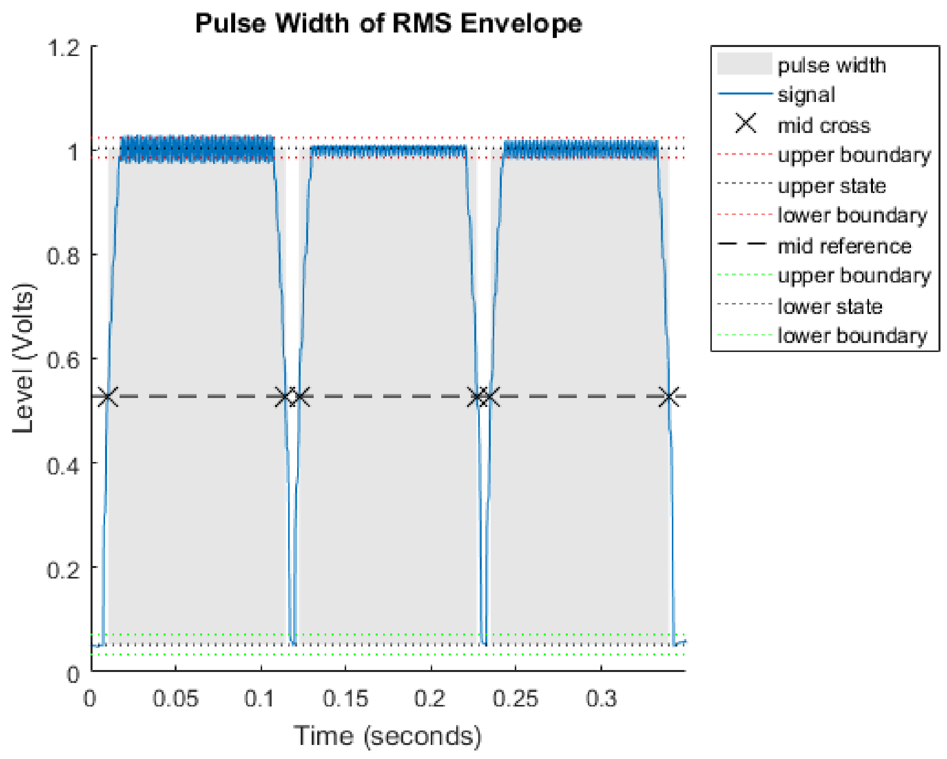


The time-domain plot of the signal confirms the presence of three bursts of energy, corresponding to three pushed buttons. To measure the length of the burst, you can take the pulse width of the RMS envelope.

```
env = envelope(tones,80,'rms');
pulsewidth(env,Fs)
title('Pulse Width of RMS Envelope')
```

```
ans =

    0.1041
    0.1042
    0.1047
```



Here you can see three pulses approximately 100 milliseconds long. However, you cannot tell which numbers were dialed. A frequency-domain plot helps you figure this out because it shows the frequencies present in the signal.

Locate the frequency peaks by estimating the mean frequency in four different frequency bands.

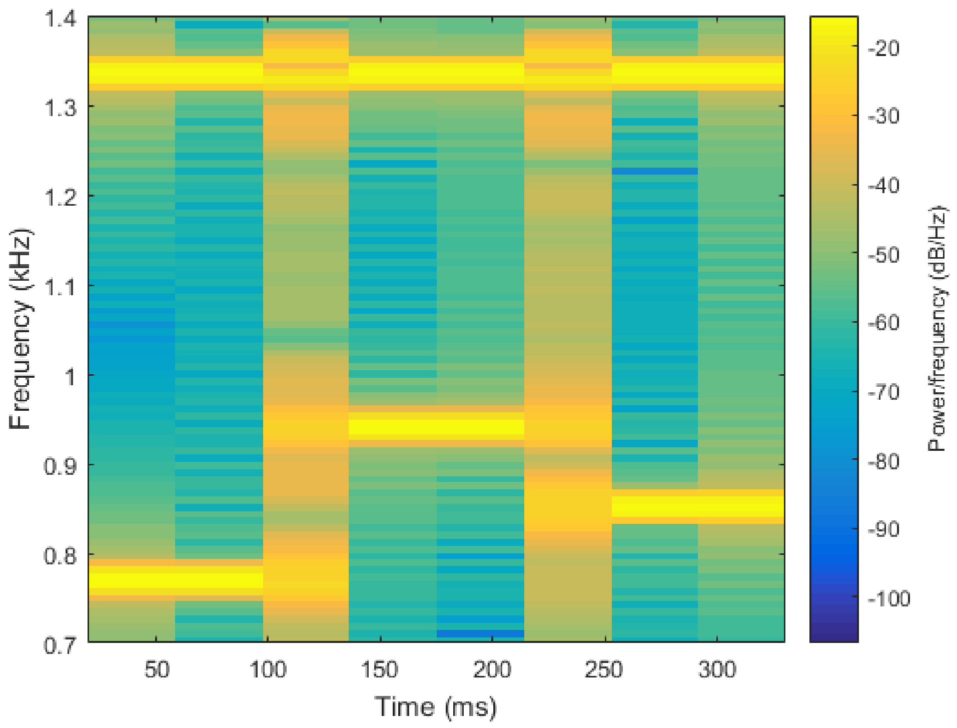
```
f = [meanfreq(tones,Fs,[700 800]), ...
     meanfreq(tones,Fs,[800 900]), ...
     meanfreq(tones,Fs,[900 1000]), ...
     meanfreq(tones,Fs,[1300 1400])];
round(f)
```

```
ans =

         770          852          941         1336
```

By matching the estimated frequencies to the diagram of the telephone pad, you can say that the dialed buttons were '5', '8', and '0'. However, the frequency-domain plot does not provide any type of time information that would allow you to figure out the order in which they were dialed. The combination could be '580','508','805','850', '085', or '058'. To solve this puzzle, use the `spectrogram` function to observe how the frequency content of the signal varies with time.

```
figure,
spectrogram(tones,[],[],[],Fs,'yaxis')
ylim([0.7 1.4])
```



The colors of the spectrogram encode frequency power levels. Yellow colors indicate frequency content with higher power; blue colors indicate frequency content with very low power. A strong yellow horizontal line indicates the existence of a tone at a particular frequency. The plot clearly shows the presence of a 1336 Hz tone in all three dialed digits, telling you that they are all on the second column of the keypad. From the plot you can see that the lowest frequency, 770 Hz, was dialed first. The highest frequency, 941 Hz, was next. The middle frequency, 852 Hz, came last. Hence, the dialed number was 508.

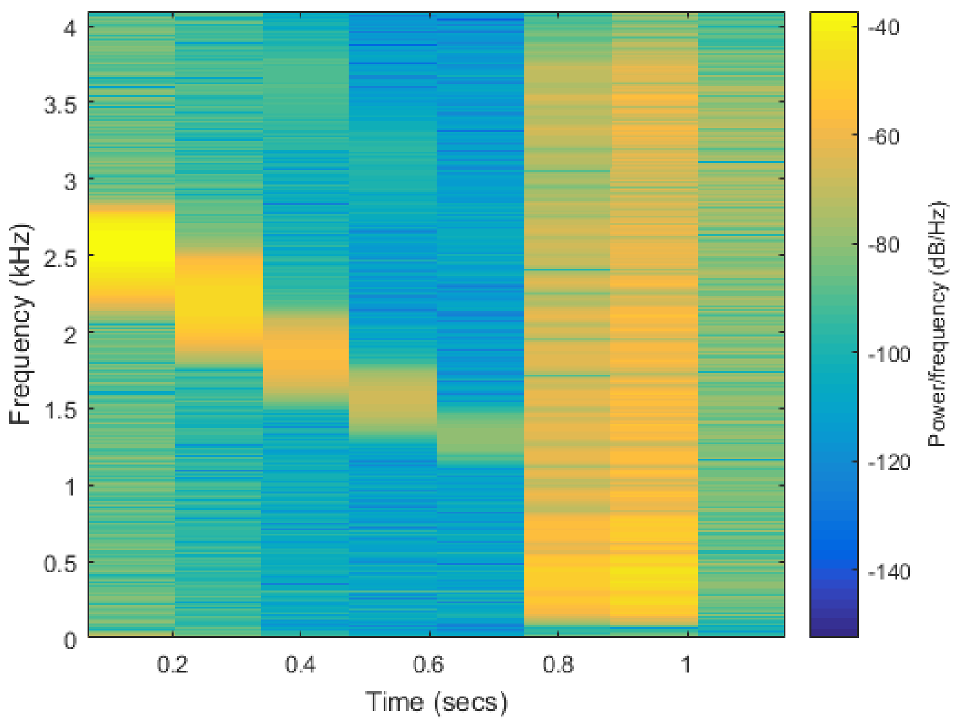
Trading Off Time and Frequency Resolution to Get the Best Representation of Your Signal

The `spectrogram` function divides a signal into segments. Longer segments provide better frequency resolution; shorter segments provide better time resolution. Consider the following recording, consisting of a chirp signal whose frequency decreases over time and a final splat sound.

```
load splat
audid = audiodevinfo(0,Fs,16,1);
if  audid ~= -1
    p = audioplayer(y,Fs,16,audid);
    play(p)
end
```

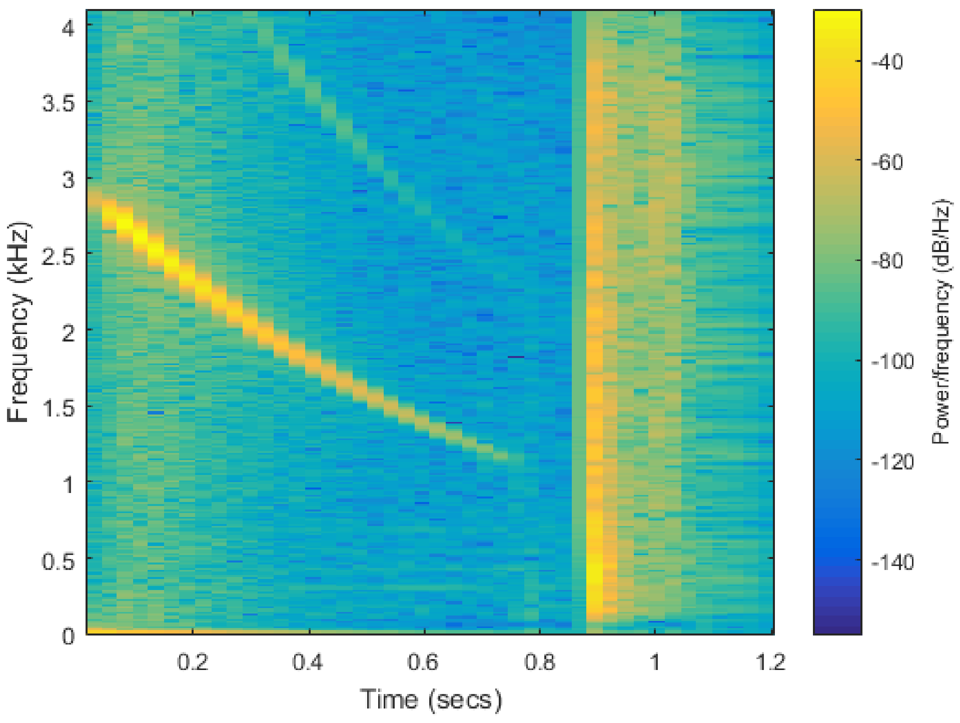
By default, the `spectrogram` function divides a real signal into segments that are approximately 22% of the length of the signal. To compute the FFT of each segment, the function overlaps the segment with 50% of the previous segment and 50% of the next. Find a window length such that `spectrogram` computes eight signal segments with 50% overlap.

```
segmentLength = round(numel(y)/4.5); % Equivalent to setting segmentLength = [] in the next line
spectrogram(y,segmentLength,[],[],Fs,'yaxis')
```



The audio and the spectrogram plot show that this signal consists of two decreasing chirps and a splat sound. The splat sound is wideband because it contains high power over the entire frequency range. By choosing a shorter segment length, you can get a better definition of the two chirps. You can also make a better estimate of the instant at which the wideband signal occurs. This instant is approximately 0.85 seconds into the recording. You can even see the brief silence between the chirps and the splat.

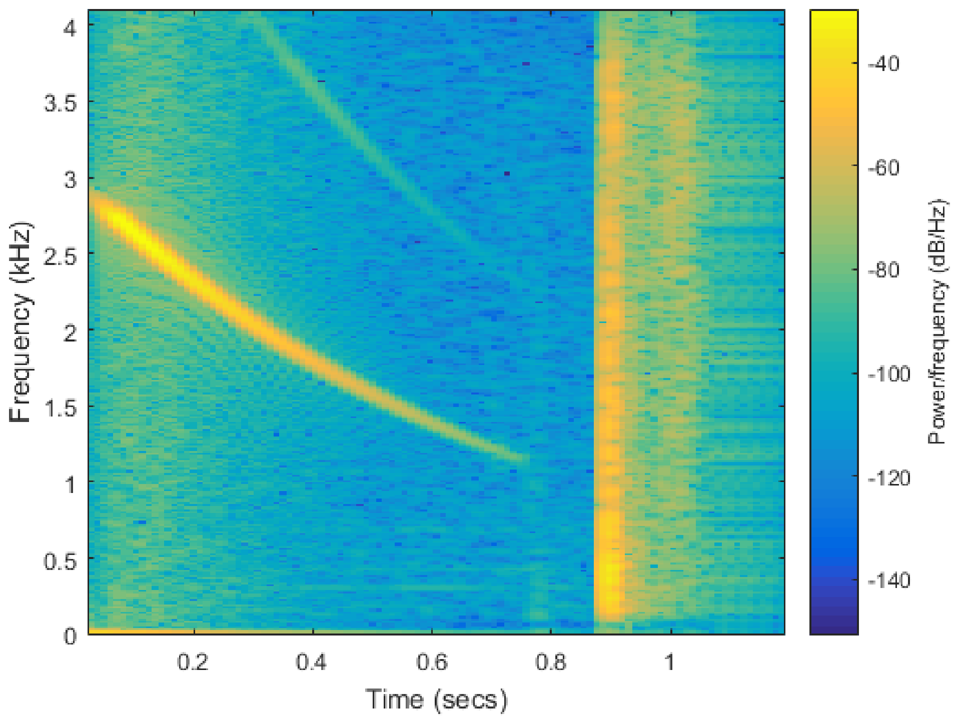
```
spectrogram(y,round(segmentLength/5),[],[],Fs,'yaxis')
```



The new plot shows that the signal had significant frequency variation over the initial segment length. In other words, the original segments were not short enough. Increasing the time resolution by decreasing the segment length allows you to resolve more time events.

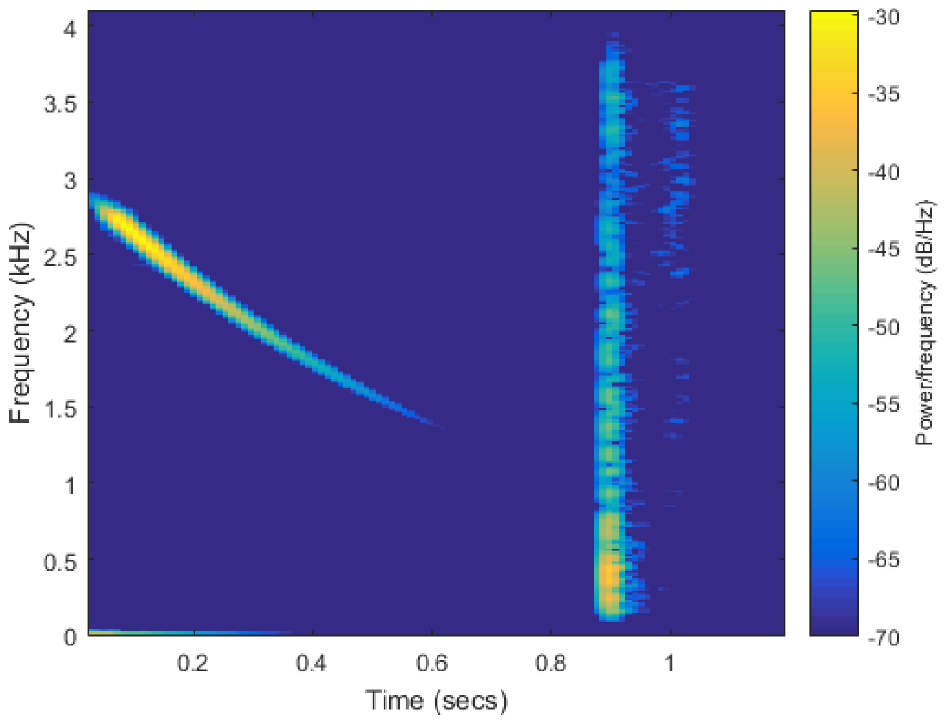
As mentioned before, the `spectrogram` function uses 50% segment overlap by default. Increasing the amount of overlap produces more spectrum lines and therefore a smoother spectrogram. Compute the spectrogram using 80% overlap to see the difference.

```
spectrogram(y,round(segmentLength/5), ...
    round(80/100*segmentLength/5),[],Fs,'yaxis')
```



To enhance prominent features of the spectrogram, you can set a minimum threshold for which estimates appear in the plot:

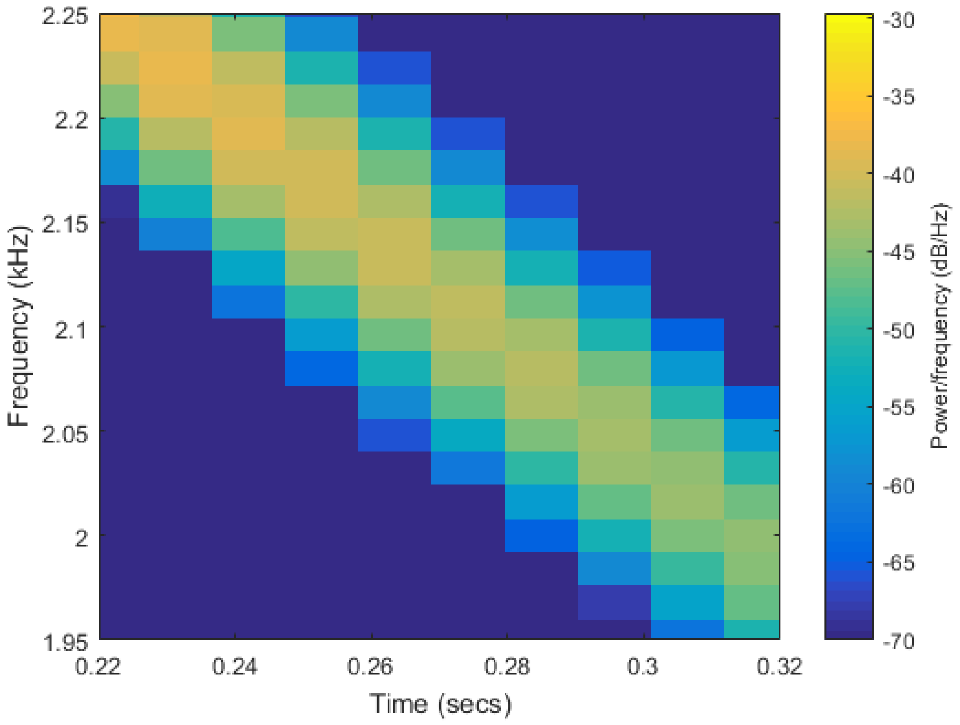
```
spectrogram(y,round(segmentLength/5), ...
    round(80/100*segmentLength/5),[],Fs,'yaxis','MinThreshold',-70)
```



Time-Frequency Reassignment

If you zoom into a portion of the chirp of our previous example, you can see that it is spread out over several adjacent frequency bins. This is due to the leakage of the windowing method used in both time and frequency.

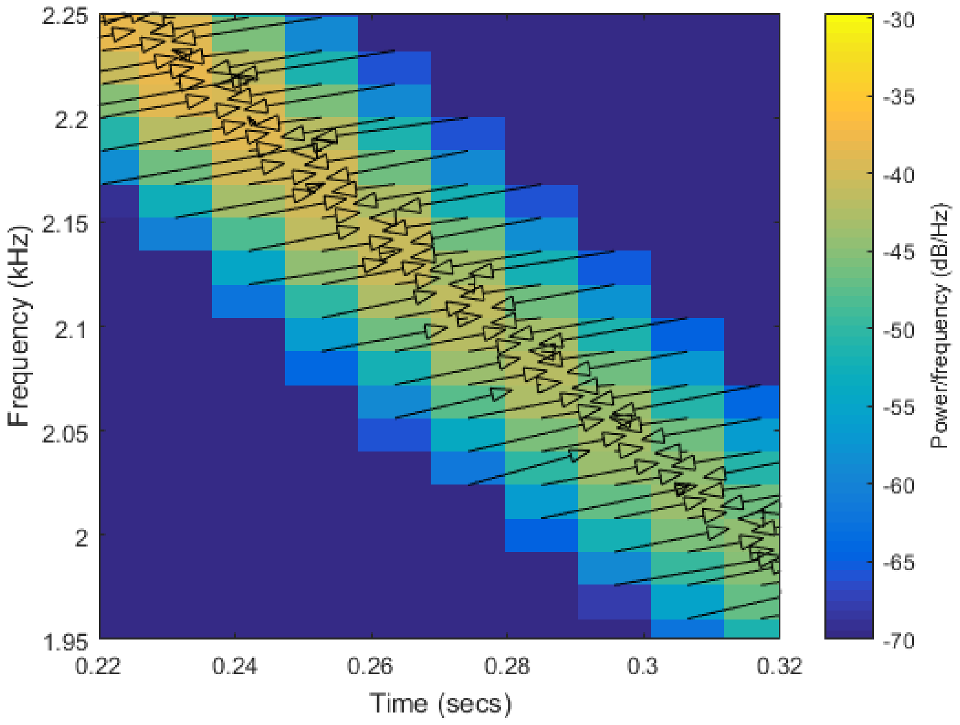
```
xlim([0.22 .32])
ylim([1.95 2.25])
```



The `spectrogram` function is capable of estimating the center of energy for each spectral estimate in both time and frequency. The arrows in the spectrogram below point to where the center estimates are located.

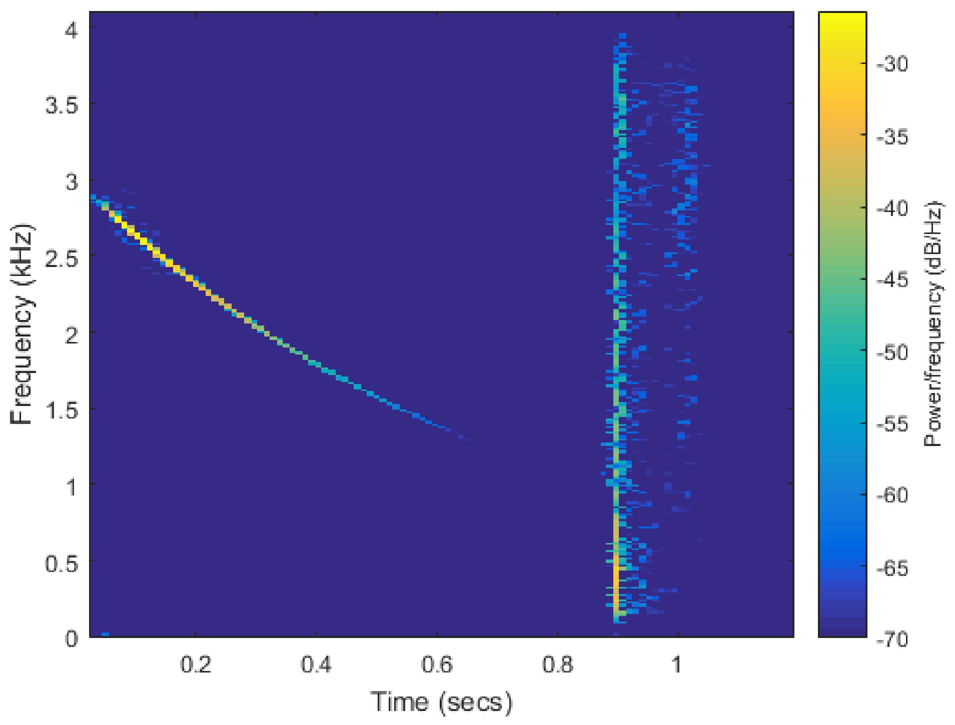
```
[~,F,T,P,Fc,Tc] = spectrogram(y,round(segmentLength/5), ...
    round(80/100*segmentLength/5),[],Fs,'yaxis','MinThreshold',-70);

helperTimeFrequencyAnalysisPlotReassignment(F,T,P,Fc,Tc,'yaxis');
```



If you reassign the energy of each estimate to the bin closest to the new time and frequency centers, you can correct for some of the leakage of the window. You can do this by using the `'reassigned'` option of the `spectrogram` function.

```
spectrogram(y,round(segmentLength/5), ...
    round(80/100*segmentLength/5),[],Fs,'yaxis','MinThreshold',-70,'reassigned')
```



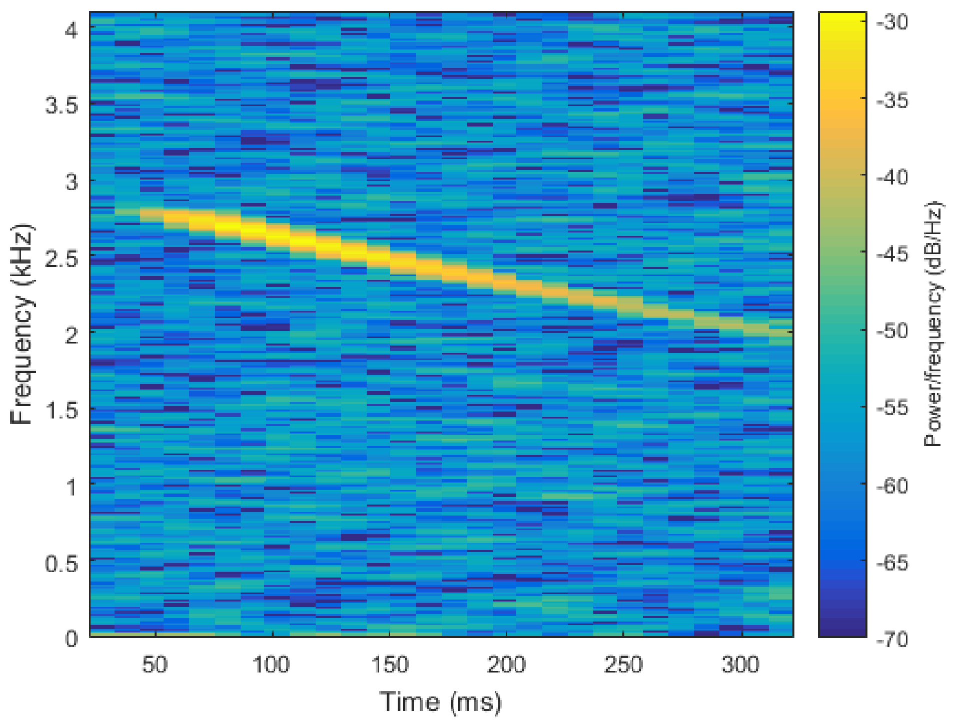
Now the chirp signal is much sharper and the "splat" sound at 0.85 seconds is better localized in time. You can also localize the signal energy using the function `fsst`, which is discussed in the next section.

Reconstructing a Time-Frequency Ridge

In this section, we reconstruct a portion of the "splat" sound by extracting a ridge in the time-frequency plane. We use `fsst` to sharpen the spectrum of a noisy version the splat signal, `tfridge` to identify the ridge of the chirp sound, and `ifsst` to reconstruct the chirp. The reconstructed signal will be denoised through this process.

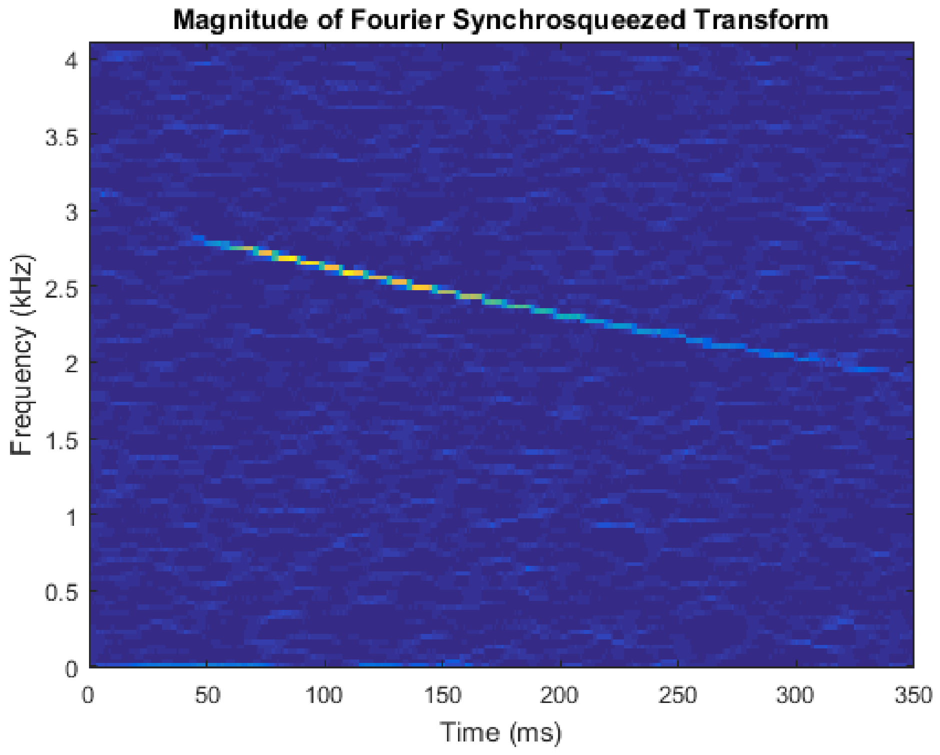
Add Gaussian noise to the chirp portion of the "splat" sound. This will simulate a noisy recording of the audio, as might occur with an inexpensive microphone. Examine the time-frequency spectral content using `spectrogram`. Use the segment length from the previous section and 80% overlap.

```
rng('default')
t = (0:length(y)-1)/Fs;
yNoise = y + 0.1*randn(size(y));
yChirp = yNoise(t<0.35);
spectrogram(yChirp,round(segmentLength/5), ...
    round(80/100*segmentLength/5),[],Fs,'yaxis','MinThreshold',-70)
```



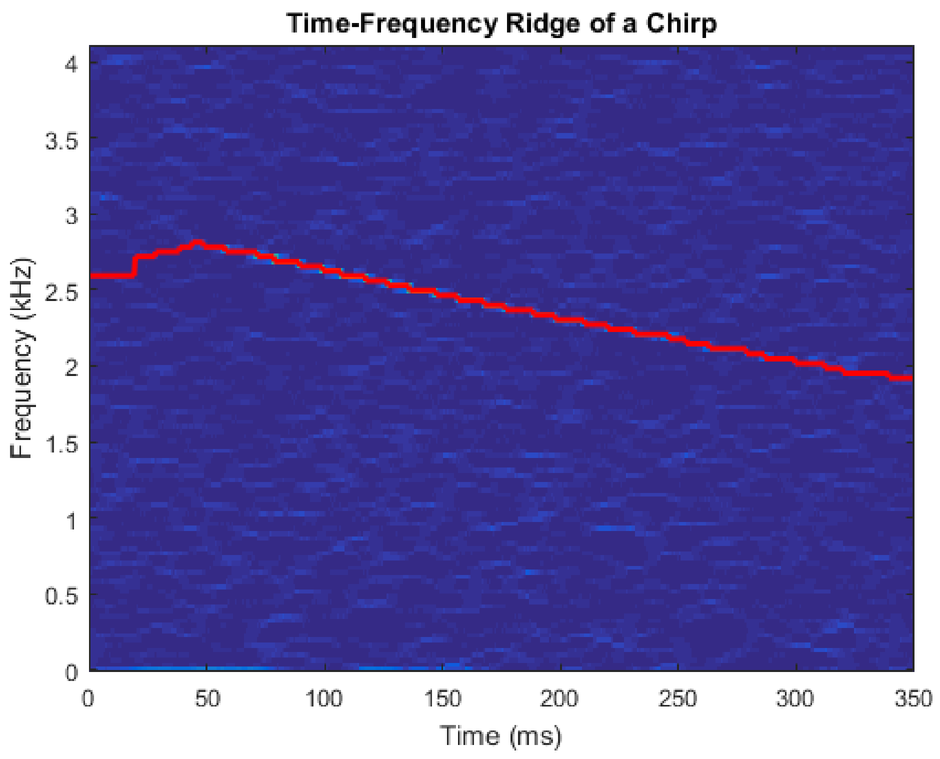
Even with a threshold, the noise energy is apparent. Next, sharpen the spectrum using the Fourier synchrosqueezed transform, `fsst`. `fsst` localizes energy in the time-frequency plane by reassigning energy in frequency for a fixed time. Compute and plot the synchrosqueezed transform of the noisy chirp.

```
fsst(yChirp,Fs,'yaxis')
```



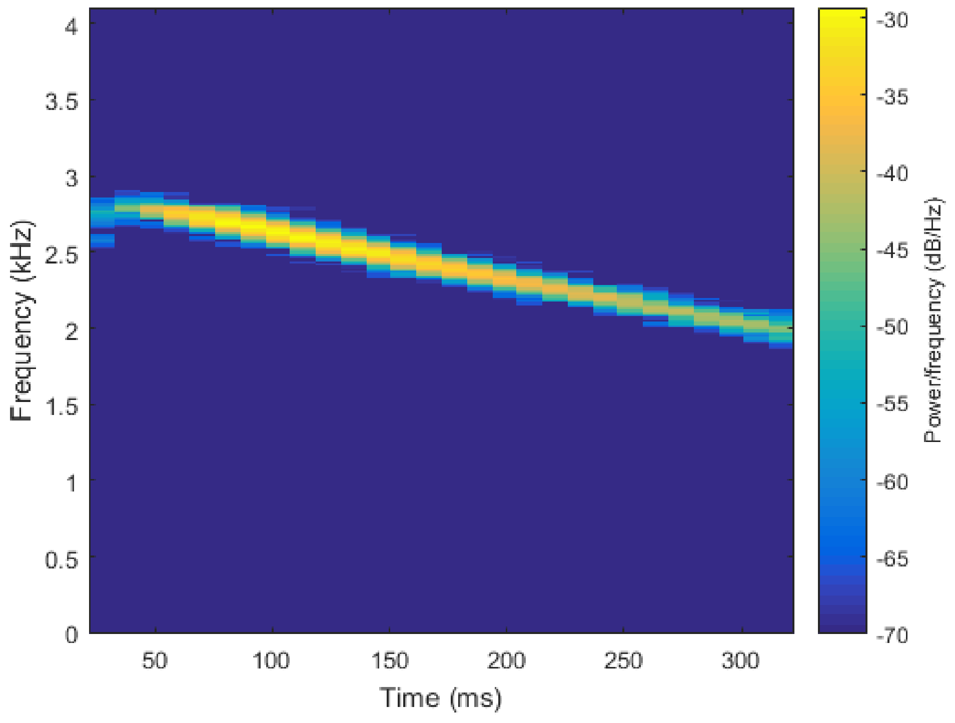
The chirp appears as a localized ridge in the time-frequency plane. Identify the ridge using `tfridge`. Plot the ridge along with the transform.

```
[sst,f] = fsst(yChirp,Fs);
[fridge, iridge] = tfridge(sst,f,10);
helperPlotRidge(yChirp,Fs,fridge);
```



Next, reconstruct the chirp signal using the ridge index vector `iridge`. Include one bin on each side of the ridge. Plot the spectrogram of the reconstructed signal.

```
yrec = ifsst(sst,kaiser(256,10),iridge,'NumFrequencyBins',1);
spectrogram(yrec,round(segmentLength/5), ...
     round(80/100*segmentLength/5),[],Fs,'yaxis','MinThreshold',-70)
```



Reconstructing the ridge has removed noise from the signal. Play the noisy and denoised signals consecutively to hear the difference.

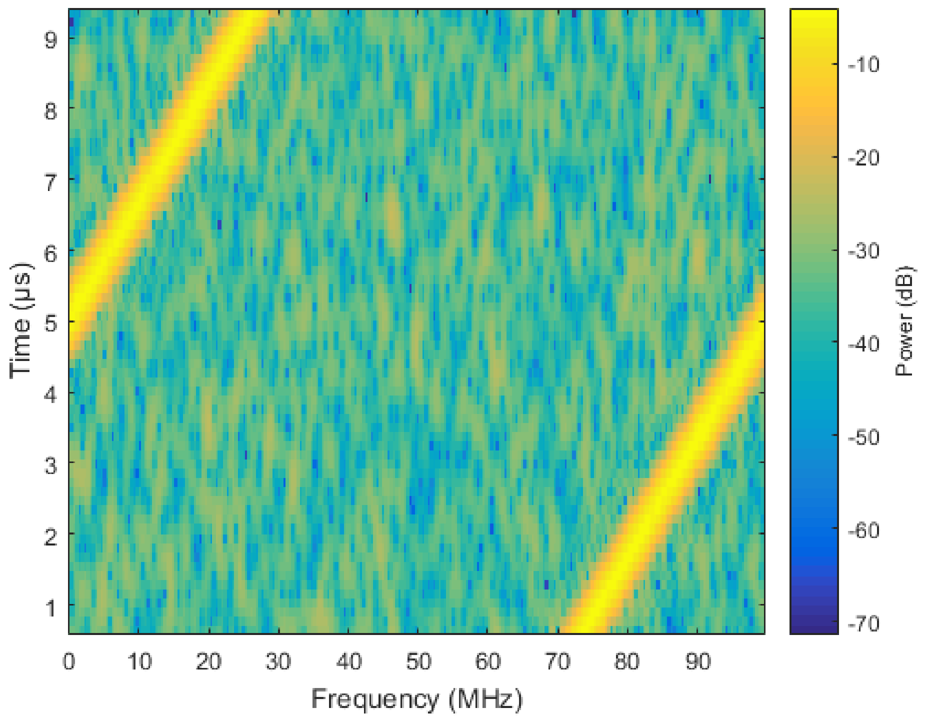
```
audid = audiodevinfo(0,Fs,16,1);
if audid ~= 01
    p = audioplayer([yChirp;zeros(size(yChirp));yrec],Fs,16,audid);
    play(p);
end
```

Measuring Power

Consider a complex linear frequency modulated (LFM) pulse, which is a common radar waveform. Plot the default two-sided power spectrogram using the `'power'` flag. Divide the signal into segments having a duration of 1.27 microseconds and use 90% overlap.

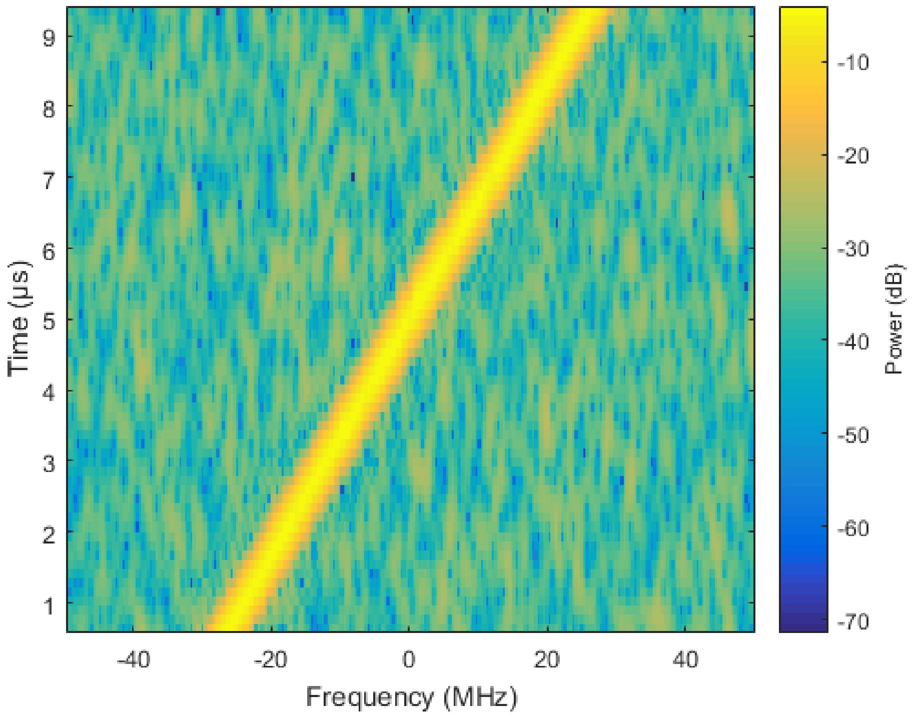
```
Fs = 1e8;
bw = 60e6;
t = 0:1/Fs:10e-6;
IComp = chirp(t,-bw/2,t(end), bw/2,'linear',90)+0.15*randn(size(t));
QComp = chirp(t,-bw/2,t(end), bw/2,'linear',0) +0.15*randn(size(t));
IQData = IComp + 1i*QComp;

segmentLength = 128;
figure
spectrogram(IQData,segmentLength, ...
    round(0.9*segmentLength),[],Fs,'power')
```



You can place the DC bin in the center of the frequency axis using the `'centered'` flag.

```
spectrogram(IQData,segmentLength, ...
    round(0.9*segmentLength),[],Fs,'power','centered')
```

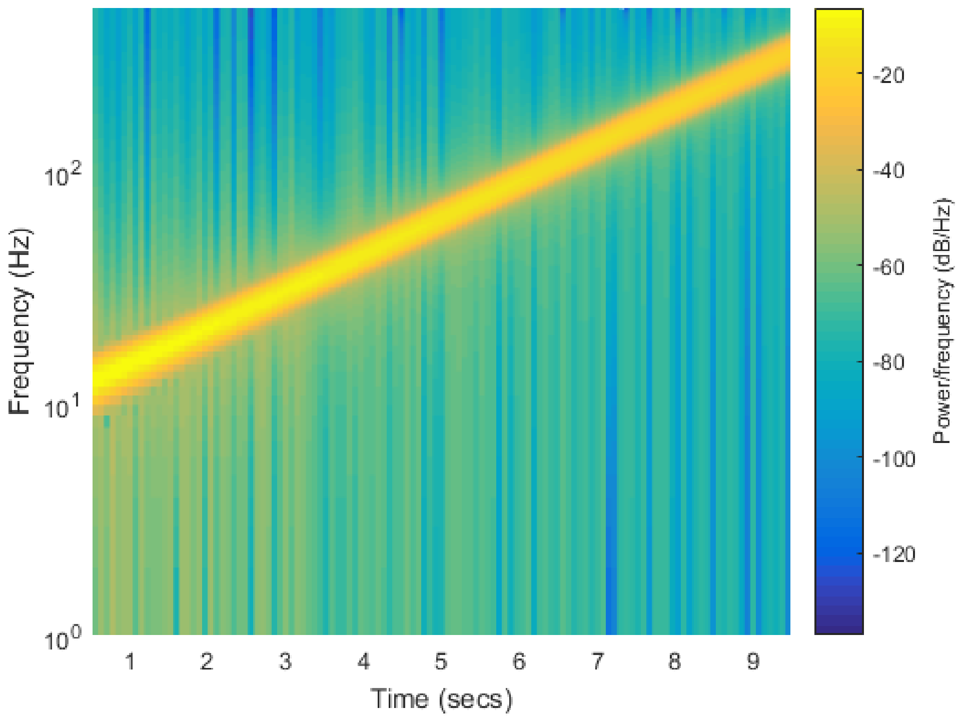


If you do not specify the `'yaxis'` flag, then the `spectrogram` function displays the frequency along the *x*-axis and the time along the *y*-axis. A segment length of 128 points (or 1.27 microseconds) with a 90% overlap gives a clear time-frequency representation of the LFM signal. The color bar shows that the power level of the signal is around -4 dB.

Logarithmic Frequency Scale Visualization

In certain applications, it may be preferable to visualize the spectrogram of a signal on a logarithmic frequency scale. You can achieve this by changing the `YScale` property of the *y*-axis. For example, consider a logarithmic chirp sampled at 1 kHz. The frequency of the chirp increases from 10 Hz to 400 Hz in 10 seconds. The spectrogram of the chirp should be represented by a straight line on a logarithmic frequency scale.

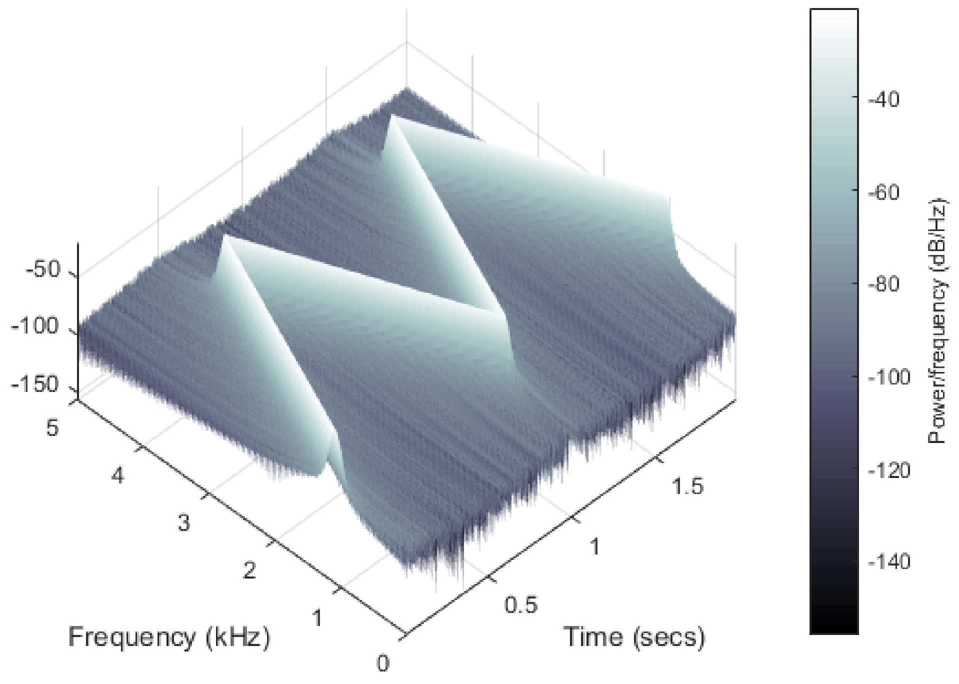
```
Fs = 1e3;
t = 0:1/Fs:10;
fo = 10;
f1 = 400;
y = chirp(t,fo,10,f1,'logarithmic');
figure
spectrogram(y,1024,950,[],Fs,'yaxis')
ax = gca;
ax.YScale = 'log';
ylim([1 Fs/2])
```



Three-Dimensional Waterfall Visualization

With the `view` command, you can visualize the spectrogram of a signal as a three-dimensional waterfall plot. You can also change the display colors with the `colormap` function.

```
Fs = 10e3;
t = 0:1/Fs:2;
x1 = vco(sawtooth(2*pi*t,0.5),[0.1 0.4]*Fs,Fs);
spectrogram(x1,kaiser(256,5),220,512,Fs,'yaxis')
view(-45,65)
colormap bone
```



Conclusions

In this example, you learned how to perform time-frequency analysis using the `spectrogram` function. You learned how to interpret spectrogram data and power levels. You learned how to change time and frequency resolution by using different sample rates. You also learned how to sharpen spectra and extract time-frequency ridges using `fsst`, `ifsst`, and `tfridge`. Finally, you learned how to configure the spectrogram plot to get a logarithmic frequency scale and three-dimensional visualization.

Appendix

The following helper functions are used in this example.

- helperDTMFToneGenerator.m
- helperTimeFrequencyAnalysisPlotReassignment.m