Sat Chirp Backgrounder

By Terry Bondy, VA3TYB

```
In [1]: printf(strftime ("Last updated: %A %e %B %Y", localtime (time ())))
```

```
Last updated: Tuesday 29 October 2019
```

Scope

What is described here is some background for a means of determining the instantaneous translating properties of a linear satellite by sending a single brief 'chirp' on the uplink and listening across the downlink satellite band with an SDR and doing some processing on the received signal. These translation properties include:

- The instantaneous translation frequency,
- Whether the received downlink signal is received USB or LSB.

But first a bit of bookkeeping to make the plots look nice.

```
In [2]: %plot --format svg
```

Producing This Document

This document is produced using Octave (https://www.gnu.org/software/octave/), an open source tool very similar to *Matlab*. The lines in the boxes are *Octave* commands used to create the mathematical examples that are used. *Octave* is being run in a Jupyter (https://jupyterlab.readthedocs.io/en/stable/) notebook which can be accessed and run from the internet at this link (https://mybinder.org/v2/gh/tbondy760/sat-chirp-notebooks/master?filepath=SatChirpBackgrounder.ipynb).

Conventions

In the *Octave* code presented, a small number of conventions are adhered to:

Case of Variables

The case of the first letter of a variable indicates if it is a scalar or a row vector, e.g.

```
In [3]: sample_rate = 20 # Sampling at 20 Hz
        T_base = [0: 1/sample_rate: 0.2] # The time base, sampling at 20 Hz
```

```
sample_rate =  20
T_base =

   0.00000   0.05000   0.10000   0.15000   0.20000
```

Matched Filter - Time Domain

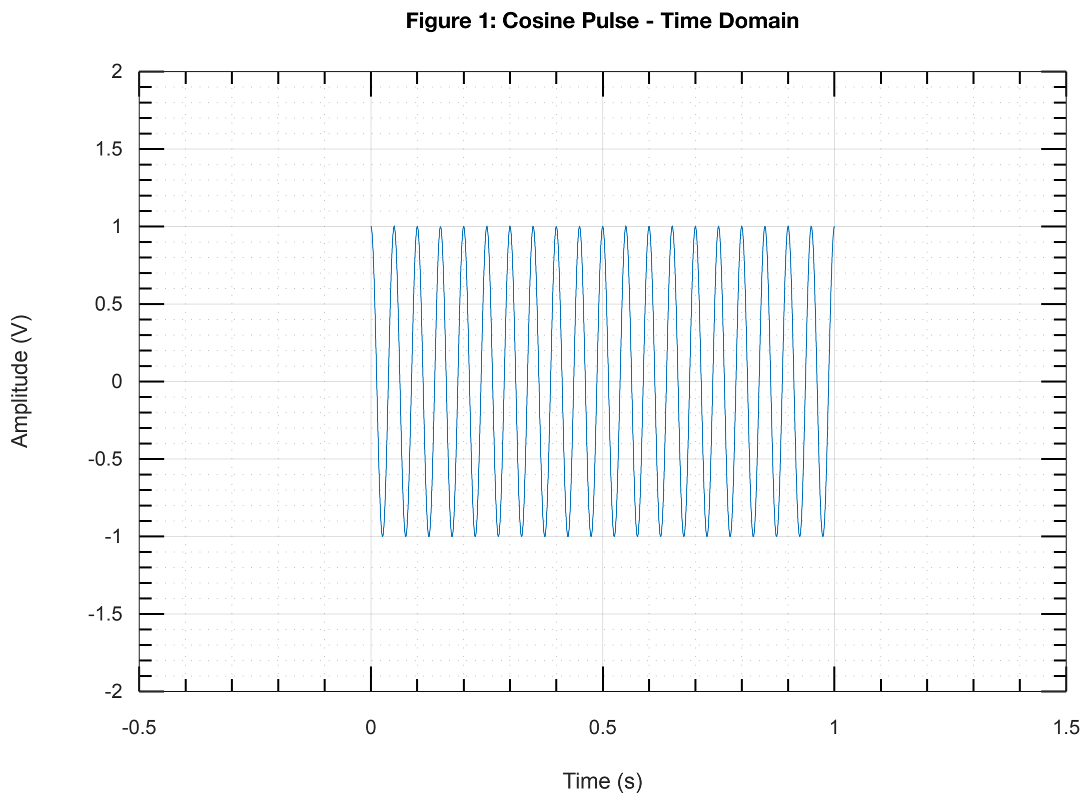
As described in a previous section, an SDR will receive the complete satellite downlink band and do processing on it while a 'chirp' is sent on the uplink. It will look at the *frequency* representation of the received chirp and determine its frequency offset from the satellite downlink band edge.

In order to determine the frequency offset, the process will use a matched filter (https://en.wikipedia.org/wiki/Matched_filter) in the *frequency* domain. Some examples of a matched filter in the time domain will demonstrate the two properties that will make it useful for this process:

- It is easy to determine whether the expected signal used to create the matched filter is present,
- It is easy to determine *when* the expected used to create the matched filter has completed.

For this example the expected signal to detect will be a 20 Hz cosine pulse 1s long (Figure 1).

In [4]:
```
sample_rate = 6000; # Sampling at 6 kHz
T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
V_expected = cos(2*pi*20*T_base);

plot(T_base, V_expected)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 1: Cosine Pulse - Time Domain")
grid on
grid minor
axis([-0.5 1.5 -2 2], "tic")
```



In the time domain, a matched filter is created by time reversing the signal to match or expected signal:

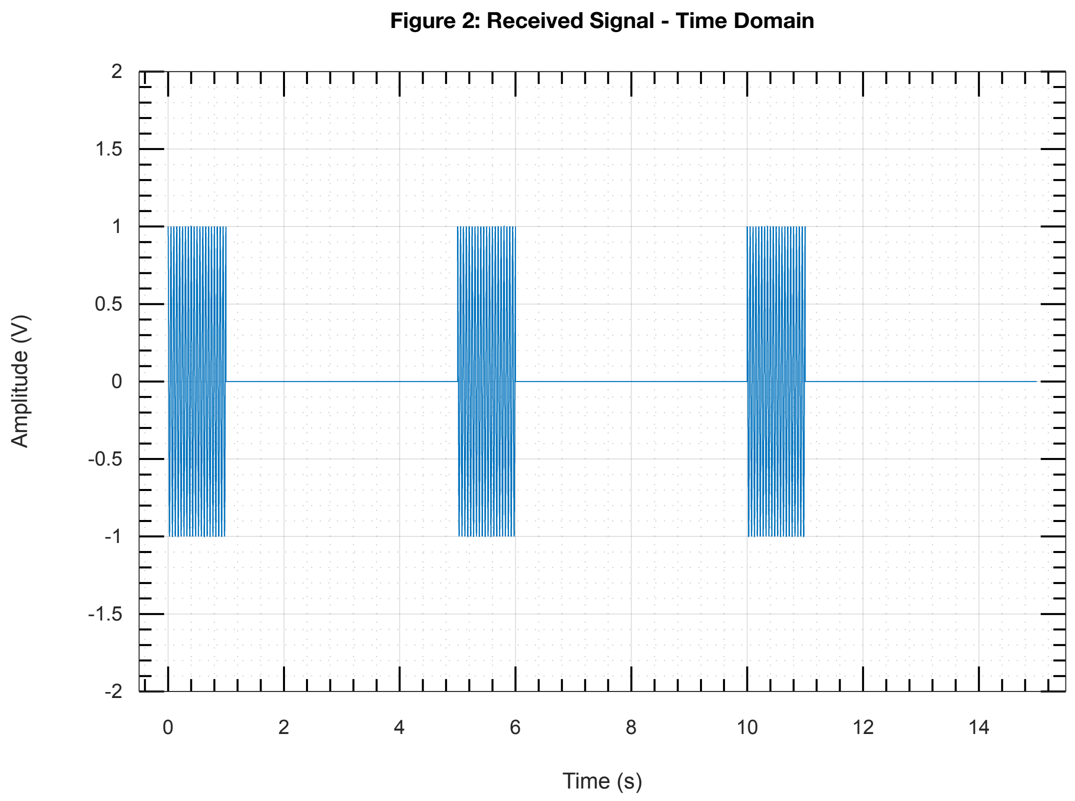
$$f_{matching}(t) = f_{expected}(-t) \quad \ldots(Eqn. 1)$$

In this case the matched filter is the signal itself because $cos(2\pi 20t) = cos(-2\pi 20t)$.

The output of the filter is determined by convolving the input signal with the matched filter.

For the first part of the example, the received signal is the expected signal itself followed by 4 sec of silence, three times (Figure 2).

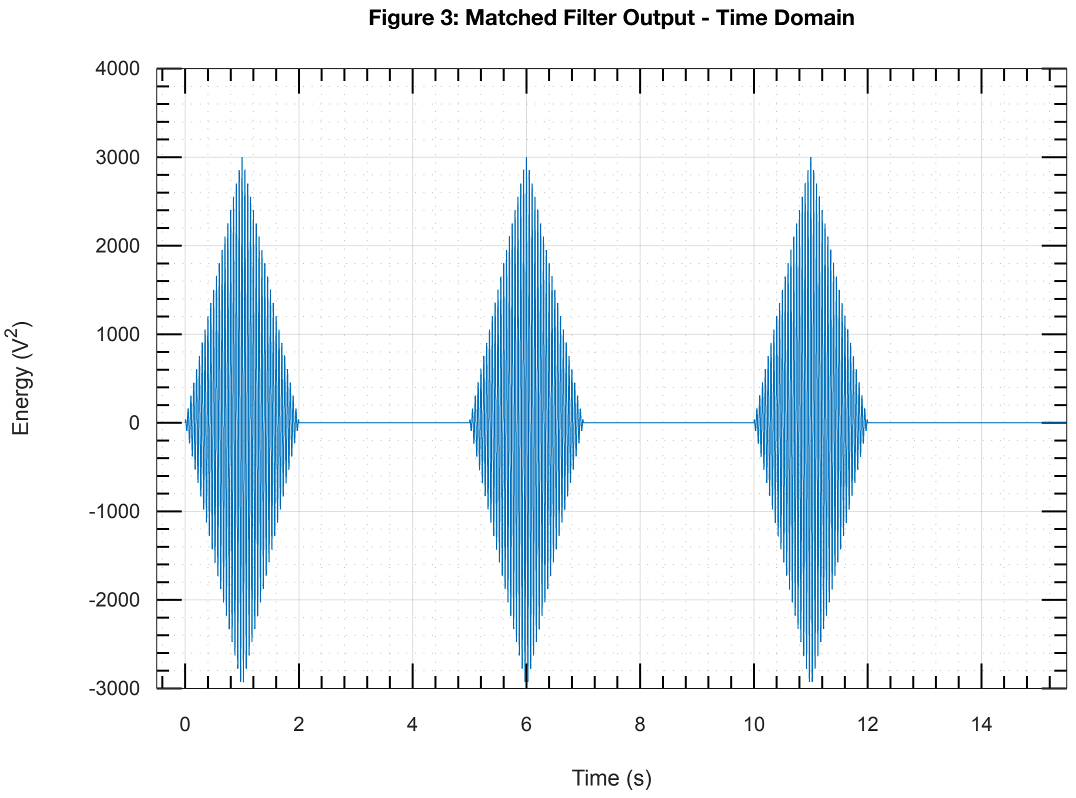
```
In [5]: V_4s_silence = zeros(1, sample_rate*4);
        V_received = horzcat(V_expected, V_4s_silence, V_expected, V_4s_silence, V_
        T_base = [0: 1/sample_rate: (size(V_received,2)-1)/sample_rate]; # The time

        plot(T_base, V_received)
        ylabel("Amplitude (V)")
        xlabel("Time (s)")
        title("Figure 2: Received Signal - Time Domain")
        grid on
        grid minor
        axis([-0.5 15.5 -2 2], "tic")
```



To do the matching the matching filter, the expected signal time reversed, is convolved with the received signal. The maximal peaks of the output indicates (Figure 3):

- Where the expected signal was detected,
- When the expected signal was completely received (i.e. the end of each pulse).

```
In [6]: V_matching = flip(V_expected); # 'flip' does the time reversing, not really
        V_matched = conv(V_matching, V_received);
        T_base = [0: 1/sample_rate: (size(V_matched,2)-1)/sample_rate]; # The time

        plot(T_base, V_matched)
        ylabel("Energy (V^2)")
        xlabel("Time (s)")
        title("Figure 3: Matched Filter Output - Time Domain")
        grid on
        grid minor
        axis([-0.5 15.5], "tic")
```



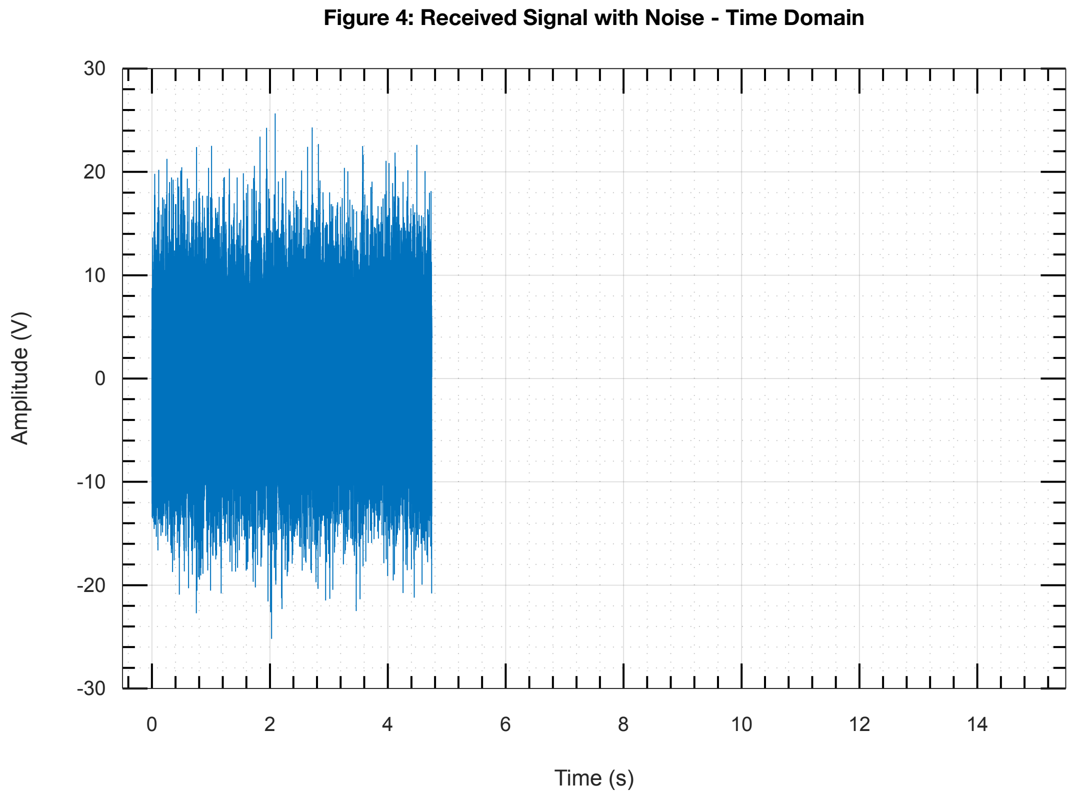
The next part of the example will show that the expected signal can be detected in the presence of a large amount of noise (i.e. SNR = -26 dB).

But first we need a function to creat some additive Gaussian white noise with energy in proportion to any signal.

In [7]:
```
#### Begin credited work
# Authored by Mathuranathan Viswanathan
# Modified by Terry Bondy, VA3TYB
# How to generate AWGN noise in Matlab/Octave by Mathuranathan Viswanathan
# is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike
# You must credit the author in your work if you remix, tweak, and build up

function Noisy_signal = add_awgn_noise(Signal, snr_dB)
    # Noise = add_awgn_noise(Signal,snr_dB) adds AWGN noise vector to signa
    # resulting signal vector Noise of specified SNR in dB
    n = length(Signal);
    snr = 10^(snr_dB/10); # SNR to linear scale
    esym = sum(abs(Signal).^2)/n; # Calculate actual symbol energy
    noise_0 = esym/snr; # Find the noise spectral density
    if(isreal(Signal)),
        noiseSigma = sqrt(noise_0); # Standard deviation for AWGN Noise whe
        Noise = noiseSigma * randn(1,n); # computed noise
    else
        noiseSigma = sqrt(noise_0/2); # Standard deviation for AWGN Noise w
        Noise = noiseSigma * (randn(1,n) + i*randn(1,n)); # computed noise
    end
    Noisy_signal = Signal + Noise;
end
#### End credited work
```

Then use it to create the received signal.

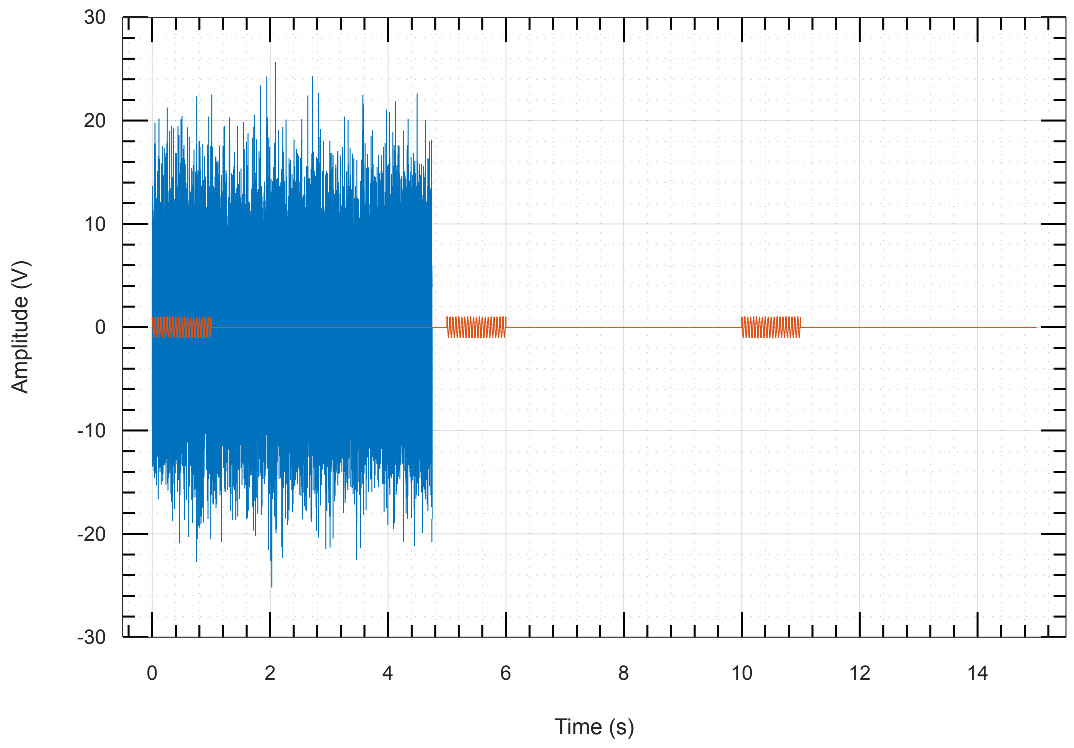
```
In [8]: snr_dB = -26;
V_noisy_received = add_awgn_noise(V_received, snr_dB);
T_base = [0: 1/sample_rate: (size(V_received,2)-1)/sample_rate]; # The time

plot(T_base, V_noisy_received)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 4: Received Signal with Noise - Time Domain")
grid on
grid minor
axis([-0.5 15.5], "tic")
```



The effect of the pulses cannot seen amongst the noise! The next plot shows with the original received signal superimposed on the noisy received for magnitude comparision (Figure 5).

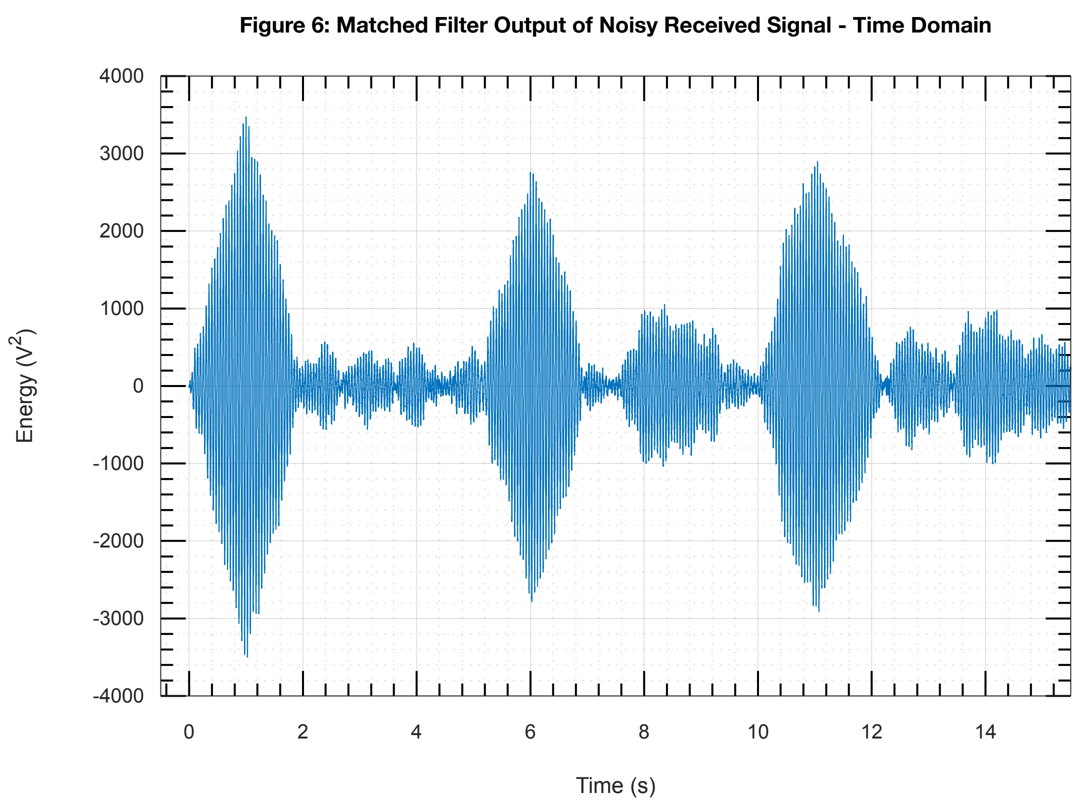
In [9]:
```
plot(T_base, V_noisy_received, T_base, V_received)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 5: Magnitude Comparision of Cosine Pulses and Noise - Time Do
grid on
grid minor
axis([-0.5 15.5], "tic")
```

Figure 5: Magnitude Comparision of Cosine Pulses and Noise - Time Domain



Examining the matched filter output (Figure 6):

```
In [10]: V_matched = conv(V_matching, V_noisy_received);
T_base = [0: 1/sample_rate: (size(V_matched,2)-1)/sample_rate]; # The time
plot(T_base, V_matched)
ylabel("Energy (V^2)")
xlabel("Time (s)")
title("Figure 6: Matched Filter Output of Noisy Received Signal - Time Doma
grid on
grid minor
axis([-0.5 15.5], "tic")
```



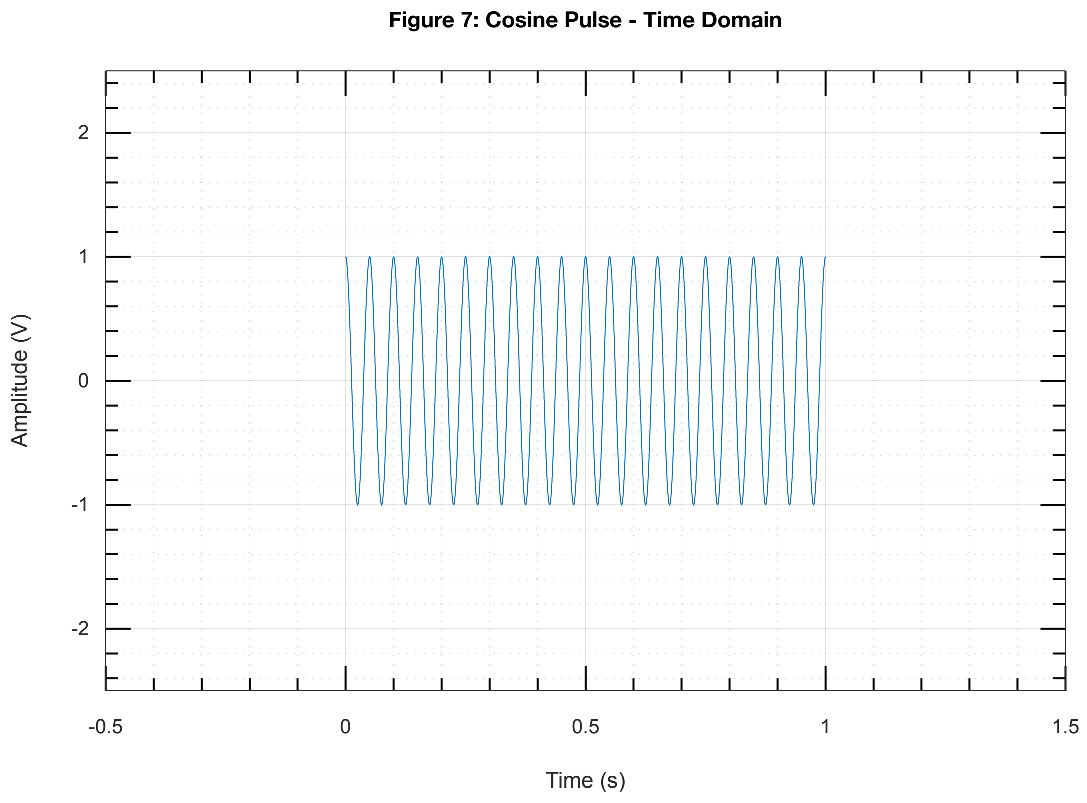
So even though the cosine pulses are 20 dB less than the noise, the matched filter is still able to detect them.

A Brief Review of Signal Analysis In The Time and Frequency Domains

Before taking on demonstrating a matched filter in the *frequency* domain, a short review of signal analysis in both the time and frequency domains is presented.

Consider again a 20 Hz cosine pulse lasting 1s (Figure 7).

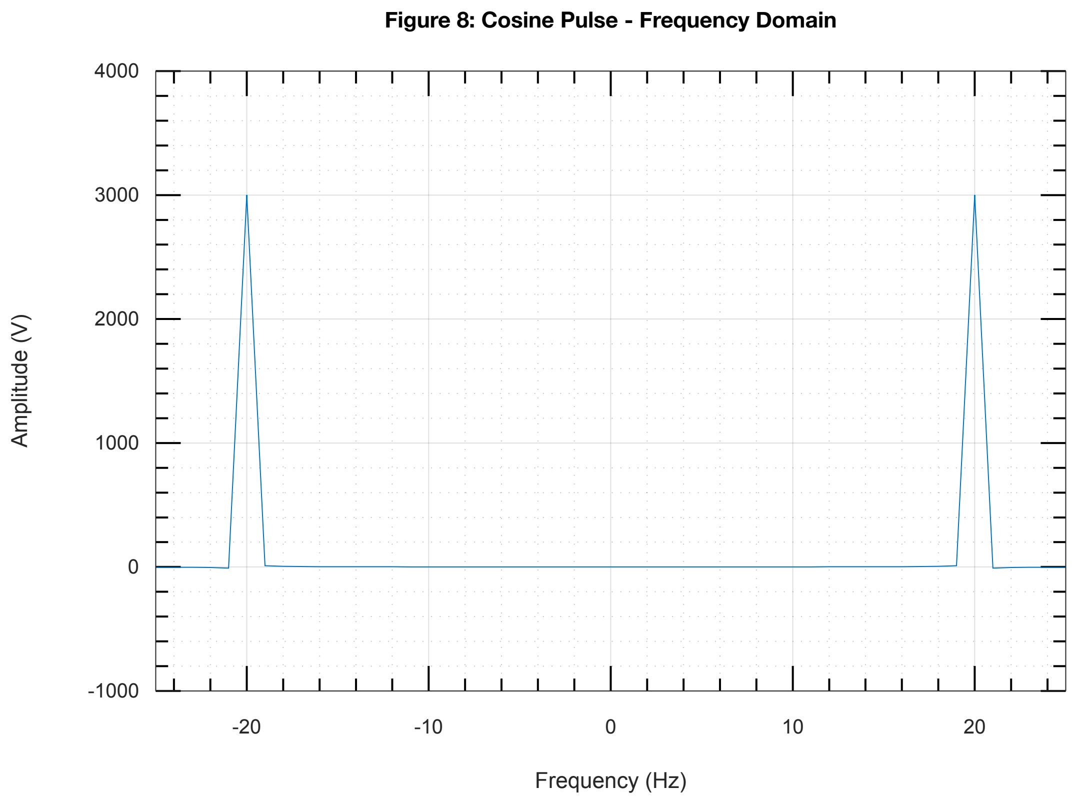
In [11]:
```
T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
V_cos_pulse = cos(2*pi*20*T_base);

plot(T_base, V_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 7: Cosine Pulse - Time Domain")
grid on
grid minor
axis([-0.5 1.5 -2.5 2.5], "tic")
```



A fast-fourier-transform (FFT) is used to determine the frequency components of the signal (Figure 8).

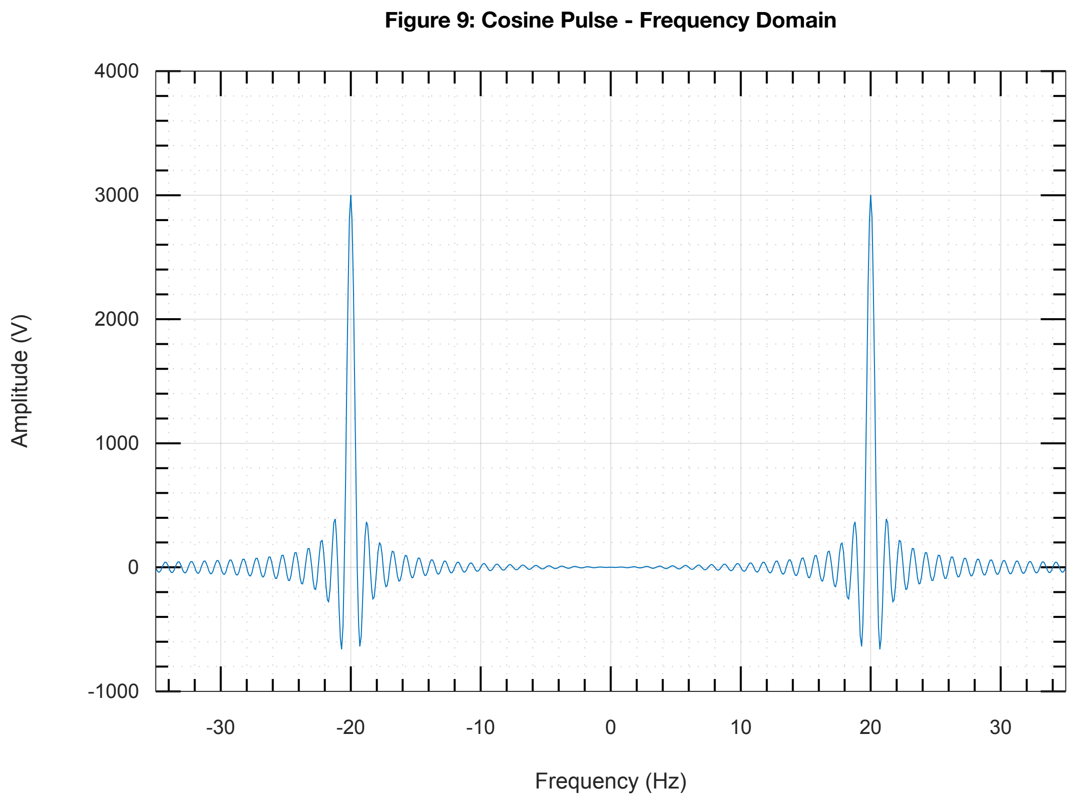
The *Octave* function `fft` is used to do the FFT. The *Octave* function `fftshift` is used to shift 0 Hz to the centre of the resulting vector of results.

In [12]:
```
E_cos_pulse = fftshift(fft(V_cos_pulse));
F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

plot(F_base, E_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Frequency (Hz)")
title("Figure 8: Cosine Pulse - Frequency Domain")
grid on
grid minor
axis([-25 25], "tic")
```



The energy at +/- 20 Hz can be seen, although the frequency resolution is lacking. To increase the resolution, the time sample can be extended. If the frequency resolution was only 1 Hz when the sample was only 1s long, it will be 0.1 Hz when the sample is 10s long (Figure 9).

```
In [13]: E_cos_pulse = fftshift(fft(V_cos_pulse, sample_rate*10 + 1));
         F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

         plot(F_base, E_cos_pulse)
         ylabel("Amplitude (V)")
         xlabel("Frequency (Hz)")
         title("Figure 9: Cosine Pulse - Frequency Domain")
         grid on
         grid minor
         axis([-35 35], "tic")
```



It can be seen from the graph:

- That the primary components consist of frequencies at and around +20 Hz and -20 Hz,
- There is "splatter" approx 15 Hz on either side of the primary +/- 20 Hz components,
- Real value (i.e. not complex, that is, $a + ib$) signals will have both +ve and -ve frequency components.

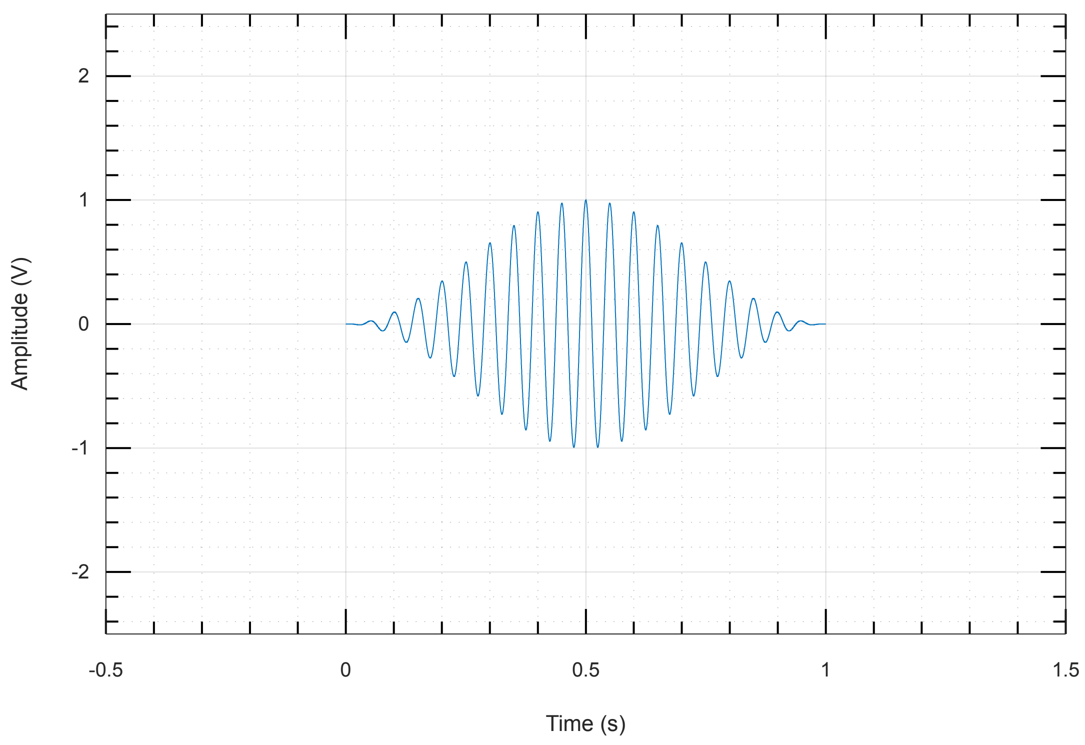
The appearance of splatter can be reasoned by considering the signal as being a continuous cosine wave modulated by an square pulse 1s long. To reduce the splatter, the continuous wave can be modulated instead with a raised cosine pulse (https://en.wikipedia.org/wiki/Raised-cosine_filter) (Figure 10). This function is called a window function (https://en.wikipedia.org/wiki/Window_function) and is used to mitigate spectral leakage. The raised cosine is also known as the Hann window (https://en.wikipedia.org/wiki/Window_function#Hann_and_Hamming_windows). The function is:

$$w(t) = \frac{1 + cos(\frac{2\pi(t-\frac{T}{2})}{T})}{2}, [\frac{-T}{2}, \frac{+T}{2}] \quad \ldots (Eqn. 2)$$

where T is the sampling period.

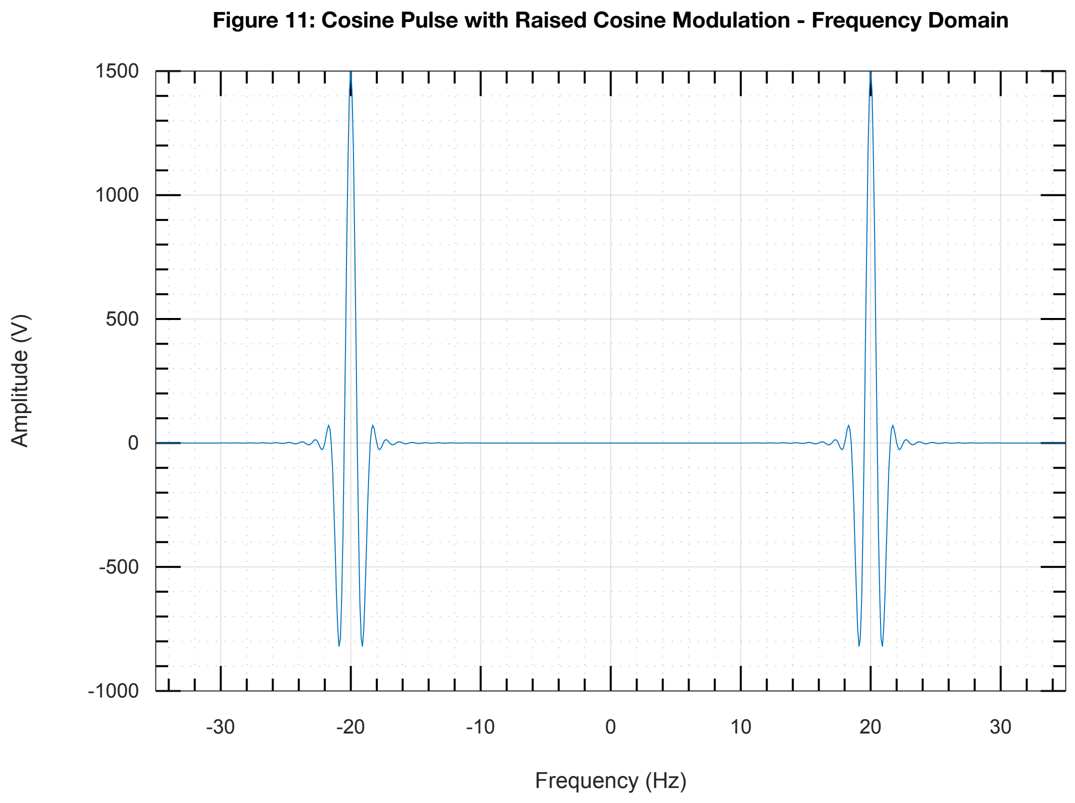
```
In [14]: T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
         V_cos_pulse = cos(2*pi*20*T_base) .* (cos(2*pi*(T_base - 0.5)) + 1)/2;

         plot(T_base, V_cos_pulse)
         ylabel("Amplitude (V)")
         xlabel("Time (s)")
         title("Figure 10: Continuous Cosine Wave with Raised Cosine Modulation - Ti
         grid on
         grid minor
         axis([-0.5 1.5 -2.5 2.5], "tic")
```

Figure 10: Continuous Cosine Wave with Raised Cosine Modulation - Time Domain



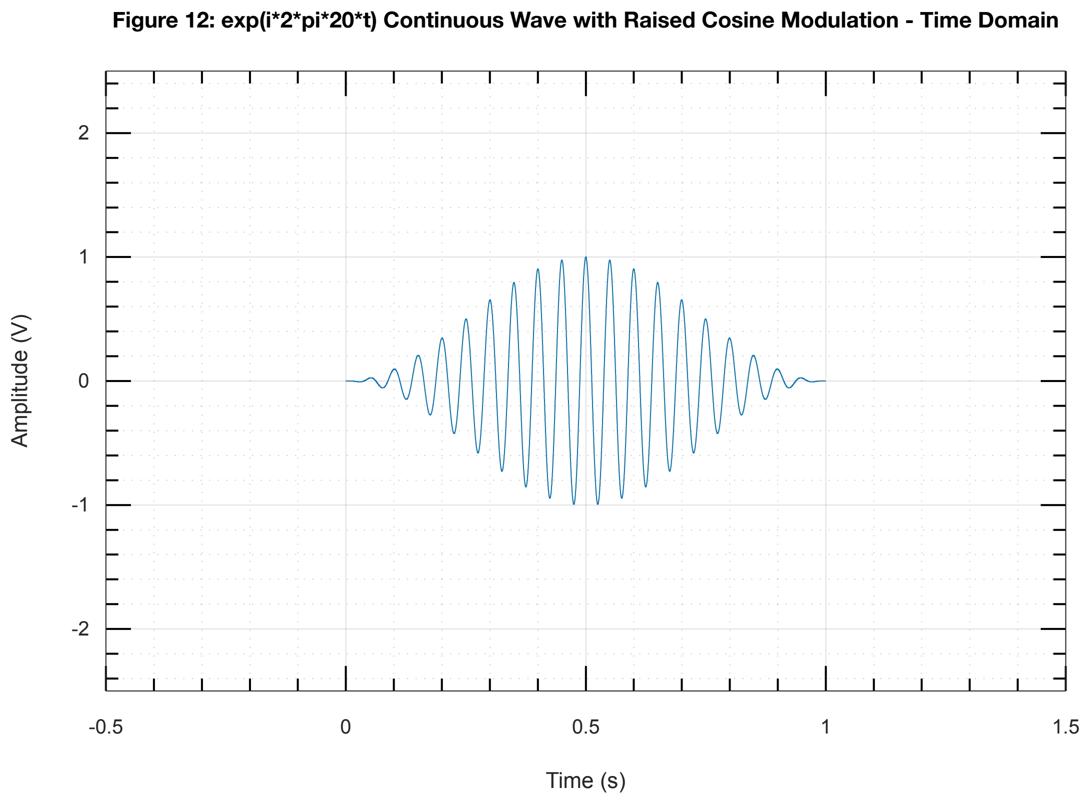
Examining this in the frequency domain (Figure 11):

In [15]:
```
E_cos_pulse = fftshift(fft(V_cos_pulse, sample_rate*10 + 1));
F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

plot(F_base, E_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Frequency (Hz)")
title("Figure 11: Cosine Pulse with Raised Cosine Modulation - Frequency Do
grid on
grid minor
axis([-35 35], "tic")
```



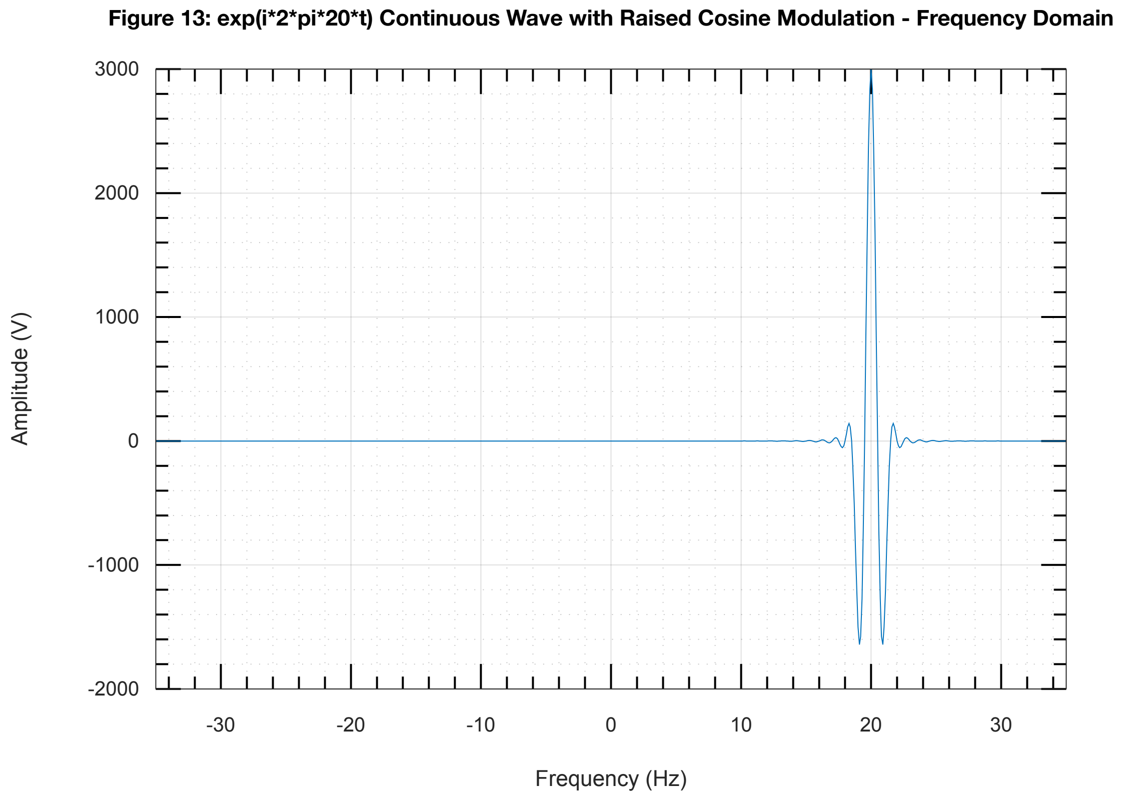
The same process can be used with complex phasors by using `exp(i*2*pi*20*T)` or `exp(-i*2*pi*20*T)` in place of `cos(2*pi*20*T)`. For the +ve phasor the time signal looks the same (Figure 12).

```
In [16]: T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
         V_cos_pulse = exp(i*2*pi*20*T_base) .* (cos(2*pi*(T_base - 0.5)) + 1)/2;

         plot(T_base, V_cos_pulse)
         ylabel("Amplitude (V)")
         xlabel("Time (s)")
         title("Figure 12: exp(i*2*pi*20*t) Continuous Wave with Raised Cosine Modul
         grid on
         grid minor
         axis([-0.5 1.5 -2.5 2.5], "tic")
```



And in the frequency domain (Figure 13):

In [17]:
```
E_cos_pulse = fftshift(fft(V_cos_pulse, sample_rate*10 + 1));
F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

plot(F_base, E_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Frequency (Hz)")
title("Figure 13: exp(i*2*pi*20*t) Continuous Wave with Raised Cosine Modul
grid on
grid minor
axis([-35 35], "tic")
```

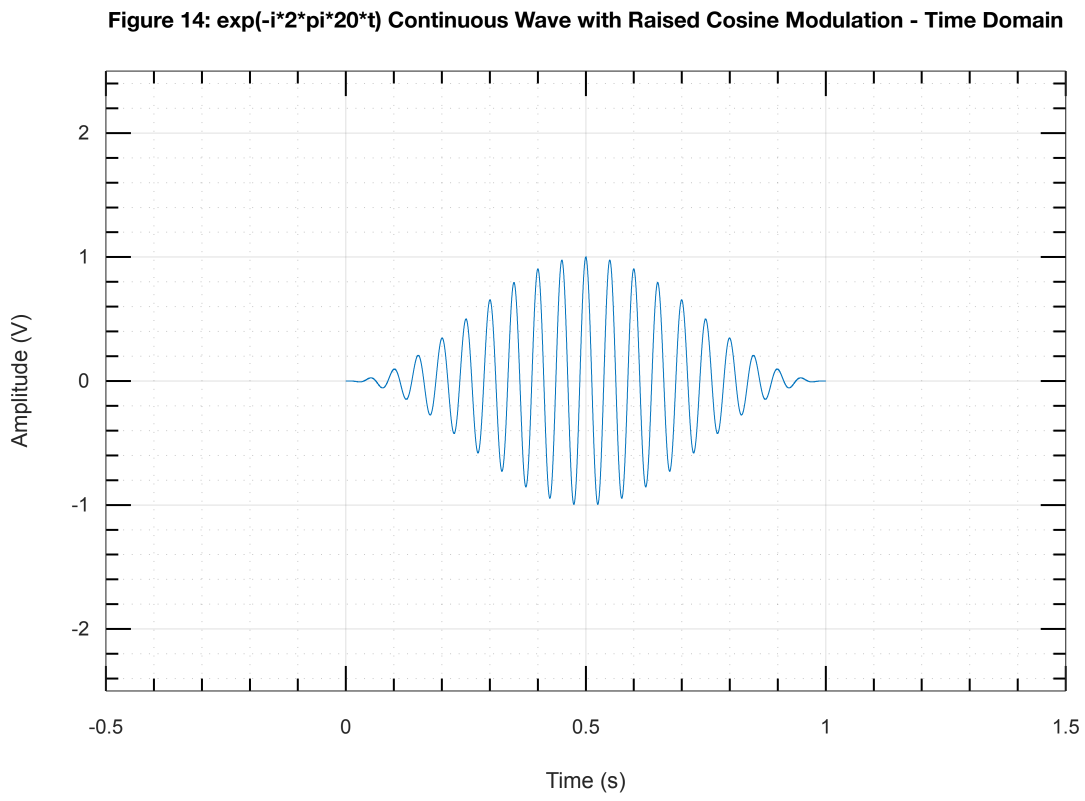


Looking at the figure it is very similar to Figure 11 except:

- That there is only one sideband, the USB,
- That the sideband has twice the energy.

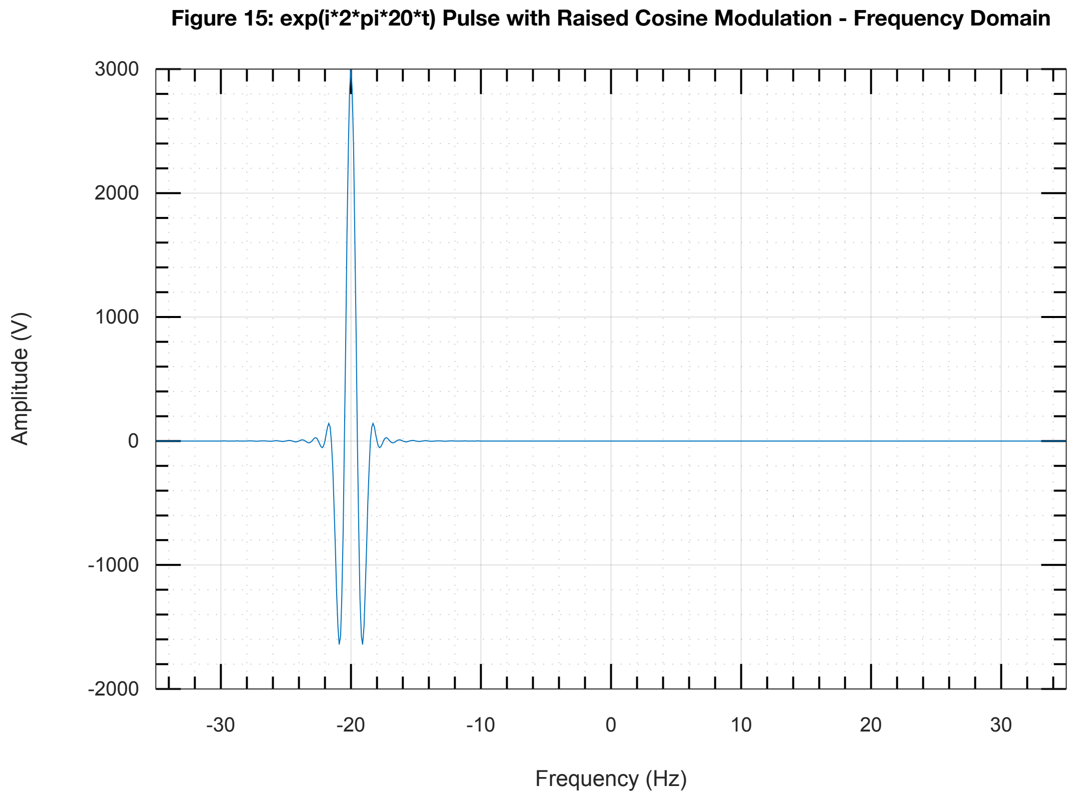
Now for `exp(-i*2*pi*20*T)` (Figure 14).

In [18]:
```
T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
V_cos_pulse = exp(-i*2*pi*20*T_base) .* (cos(2*pi*(T_base - 0.5)) + 1)/2;

plot(T_base, V_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 14: exp(-i*2*pi*20*t) Continuous Wave with Raised Cosine Modu
grid on
grid minor
axis([-0.5 1.5 -2.5 2.5], "tic")
```



It looks identical to Figure 10 and Figure 12. And in the frequency domain (Figure 15):

```
In [19]: E_cos_pulse = fftshift(fft(V_cos_pulse, sample_rate*10 + 1));
         F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

         plot(F_base, E_cos_pulse)
         ylabel("Amplitude (V)")
         xlabel("Frequency (Hz)")
         title("Figure 15: exp(i*2*pi*20*t) Pulse with Raised Cosine Modulation - Fr
         grid on
         grid minor
         axis([-35 35], "tic")
```



Looking at the figure it is very similar to Figure 11 except:

- That there is only one sideband, the LSB,
- That the sideband has twice the energy.

The following trig identity has also been indirectly demonstrated:

$$cos(kt) = \frac{e^{ikt} + e^{-ikt}}{2}$$

Matched Filter In The Frequency Domain

So far in the examples a matched filter is shown working in the time domain. Matched filters can also be used in the frequency domain. In this case it is possible to determine the frequency offset of the signal. The process is the following:

- Expected signal $\rightarrow$ FFT $\rightarrow$ flip in frequency = matched filter
- Received signal $\rightarrow$ FFT $\rightarrow$ convolve with matched filter = matched filter output

There is a property in time/frequency analysis that can be brought to bear, so that performing convolution can be avoid: convolution of two signals in one domain is multiplication in the other. So now the process becomes:

- Expected signal → FFT → flip in frequency → reverse FFT → matched filter transformed to time domain
- Received signal → multiply by matched filter transformed to time domain → FFT = matched filter output

This example will use the pulse from Figure 12 again, namely

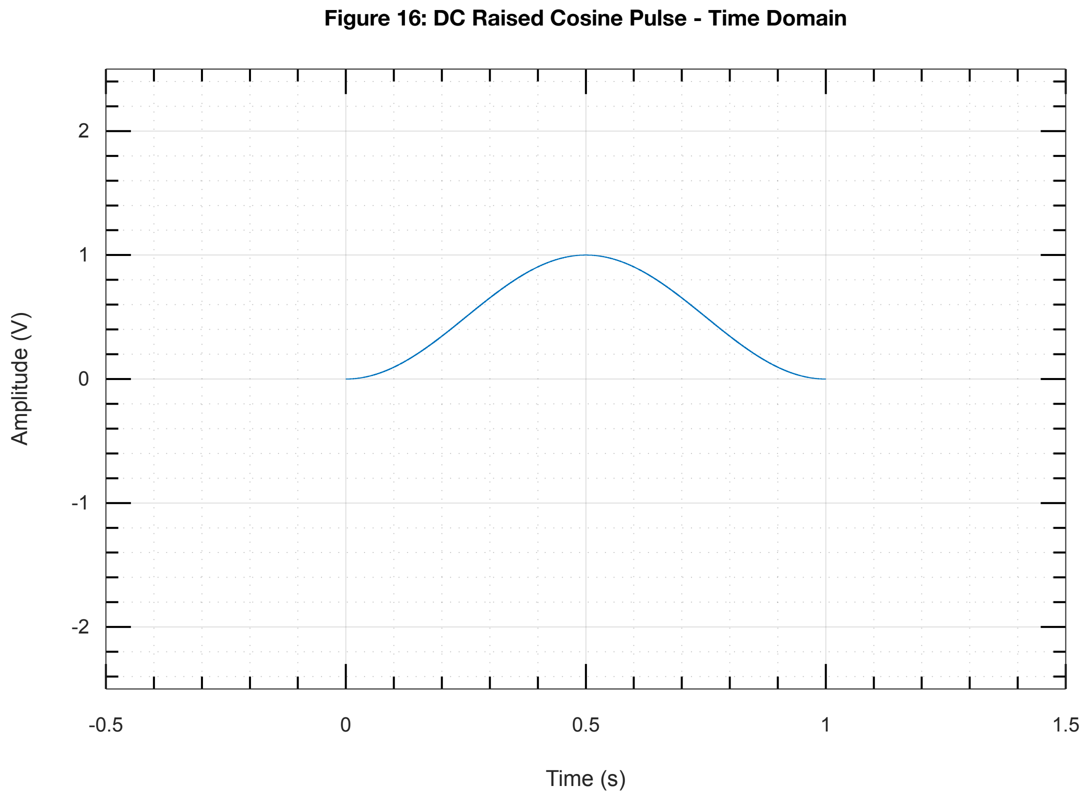
```
V_cos_pulse = exp(i*2*pi*20*T_base) .* (cos(2*pi*(T_base - 0.5)) + 1)/2
```

And when it occurs, a 0 offset should be read from the frequency graph. In order for it to do this, the matched signal should be proportional to the DC raised cosine pulse:

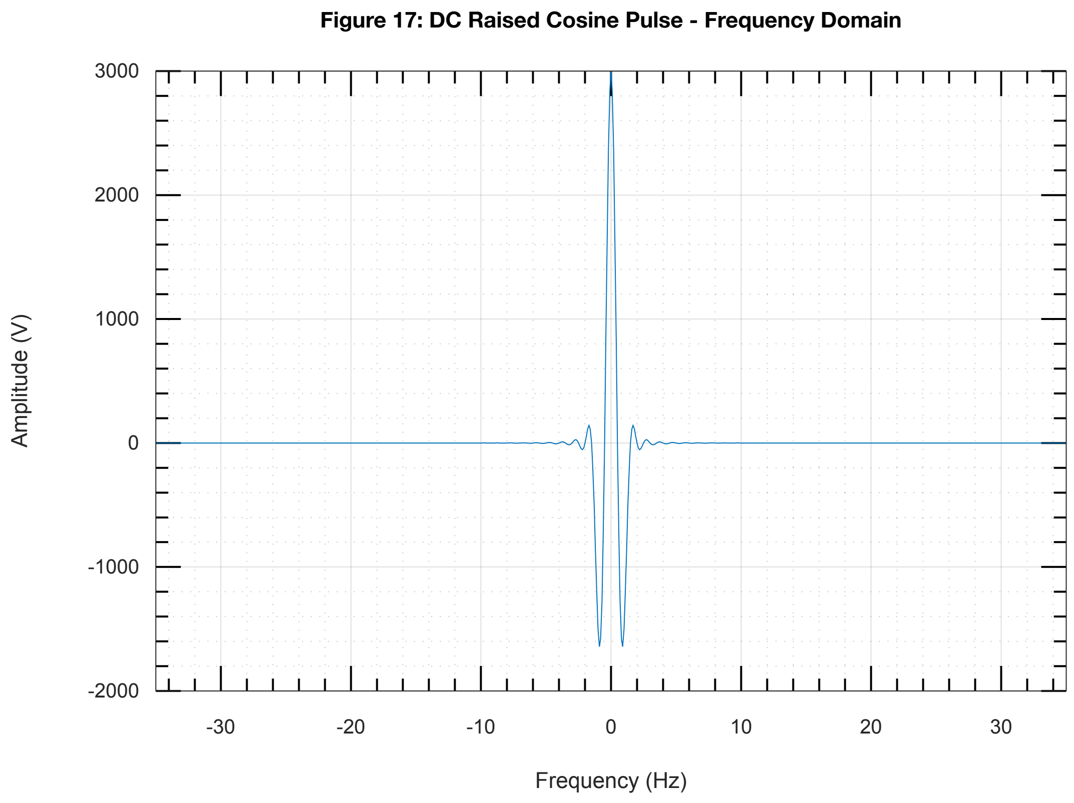
$$\frac{1 + cos(2\pi(t - 0.5))}{2}, -0.5 <= t <= 0.5$$

Examine the function and its frequency response (Figure 16 & Figure 17)

In [20]:
```
T_base = [0: 1/sample_rate: 1]; # The time base, sampling at 6 kHz
V_cos_pulse = (cos(2*pi*(T_base - 0.5)) + 1)/2;

plot(T_base, V_cos_pulse)
ylabel("Amplitude (V)")
xlabel("Time (s)")
title("Figure 16: DC Raised Cosine Pulse - Time Domain")
grid on
grid minor
axis([-0.5 1.5 -2.5 2.5], "tic")
```



```
In [21]: E_cos_pulse = fftshift(fft(V_cos_pulse, sample_rate*10 + 1));
         F_base = linspace(-sample_rate/2, sample_rate/2, size(E_cos_pulse,2));

         plot(F_base, E_cos_pulse)
         ylabel("Amplitude (V)")
         xlabel("Frequency (Hz)")
         title("Figure 17: DC Raised Cosine Pulse - Frequency Domain")
         grid on
         grid minor
         axis([-35 35], "tic")
```



But the DC raised cosine pulse is what should result *after* the multiplication of the expected input signal with the sampling function, or:

$$V_{matched}(t) = \frac{cos(2\pi(t - 0.5)) + 1}{2} = V_{expectedinput}(t)V_{matching}(t)$$

If

$$V_{matching}(t) = \frac{cos(2\pi(t - 0.5)) + 1}{2}V'_{matching}(t)$$

then

$$1 = V_{expectedinput}(t)V'_{matching}(t)$$

or

$$V'_{matching}(t) = \frac{1}{V_{expectedinput}(t)}$$

If the pulse from Figure 12 is going to be used as the expected input, or

```
V_cos_pulse = exp(i*2*pi*20*T_base) .* (cos(2*pi*(T_base - 0.5)) +
 1)/2
```

then

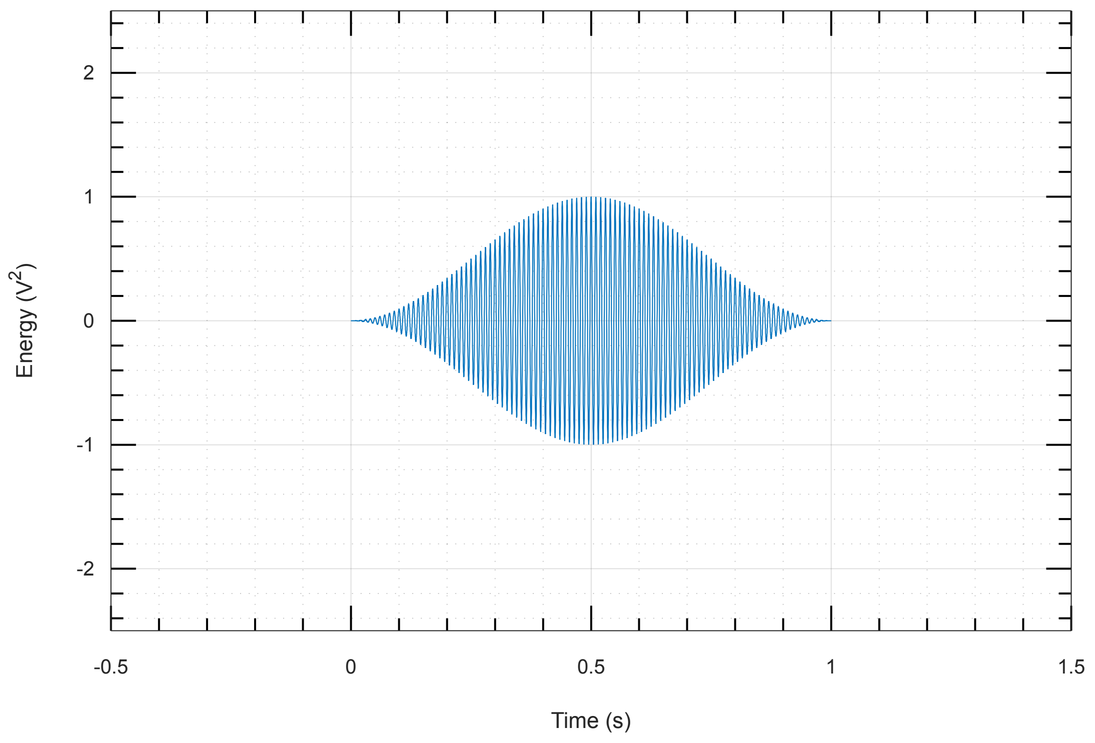
```
V_matching = 1/(exp(i*2*pi*20*T_base)) .* (cos(2*pi*(T_base - 0.5))
+ 1)/2
           = exp(-i*pi*20*T_base)) .* (cos(2*pi*(T_base - 0.5)) +
 1)/2
```

which corresponds to the expected signal's reversal in the frequency domain.

Attempt this with a signal at 120 Hz, or the same as the expected input signal but shifted up 100 Hz, and see what the time and frequency graphs indicate (Figure 18 and Figure 19 respectively).

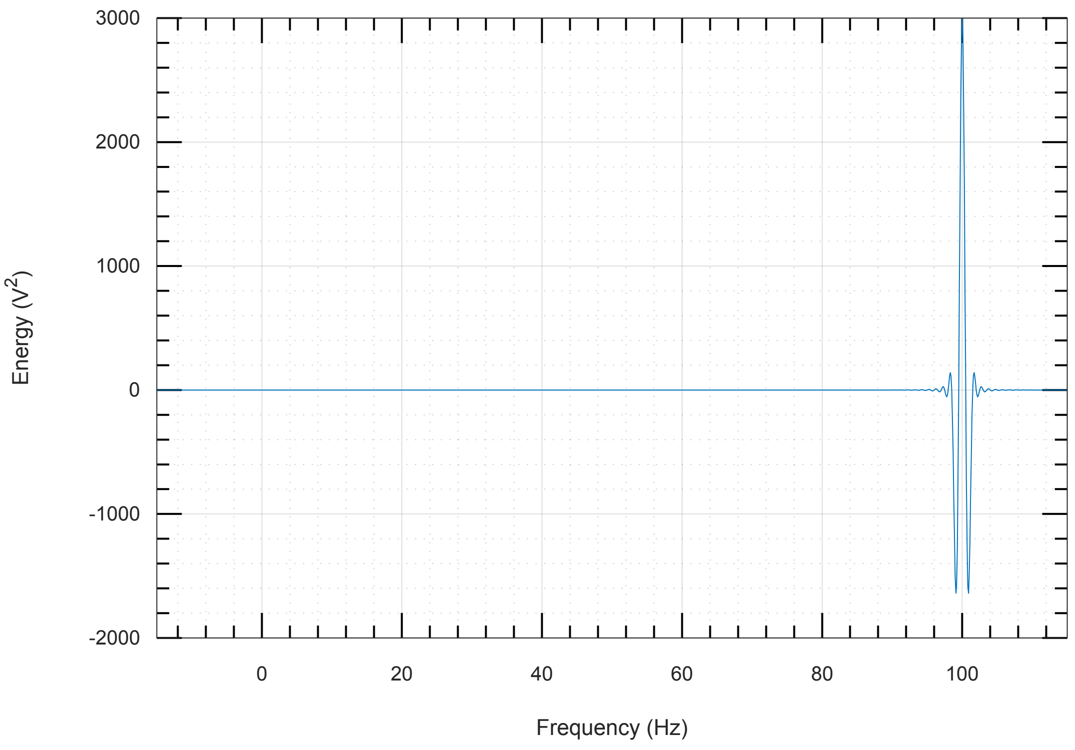
In [22]:
```
V_matching = 1 ./ (exp(i*2*pi*20*T_base)) .* (cos(2*pi*(T_base - 0.5)) + 1)
V_new_signal = exp(i*2*pi*120*T_base); # Notice the 120 term
# Multiplication in the time domain instead of convolution in the frequency
V_matched = V_matching .* V_new_signal;

plot(T_base, V_matched)
ylabel("Energy (V^2)")
xlabel("Time (s)")
title("Figure 18: Matched Signal When Expected Signal 100 Hz Up - Time Doma
grid on
grid minor
axis([-0.5 1.5 -2.5 2.5], "tic")
```

Figure 18: Matched Signal When Expected Signal 100 Hz Up - Time Domain



```
In [23]: E_matched = fftshift(fft(V_matched, sample_rate*10 + 1));
         F_base = linspace(-sample_rate/2, sample_rate/2, size(E_matched,2));

         plot(F_base, E_matched)
         ylabel("Energy (V^2)")
         xlabel("Frequency (Hz)")
         title("Figure 19: Matched Signal When Expected Signal 100 Hz Up - Frequency
         grid on
         grid minor
         axis([-15 115], "tic")
```

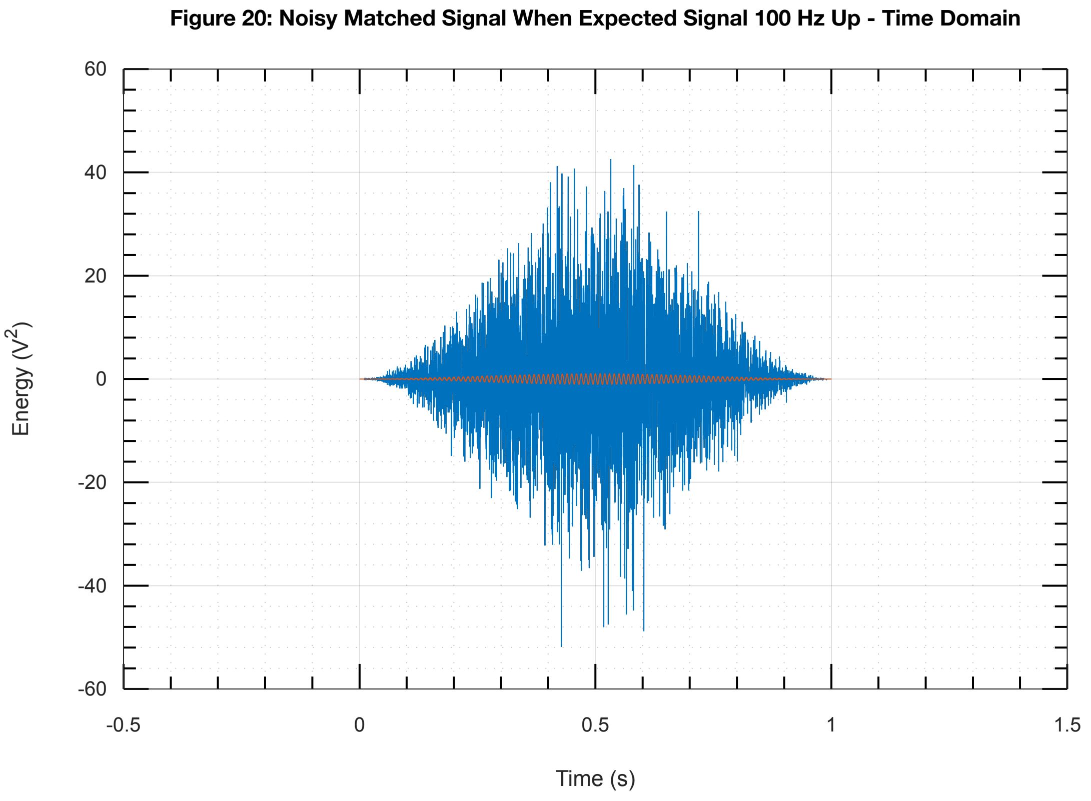
Figure 19: Matched Signal When Expected Signal 100 Hz Up - Frequency Domain



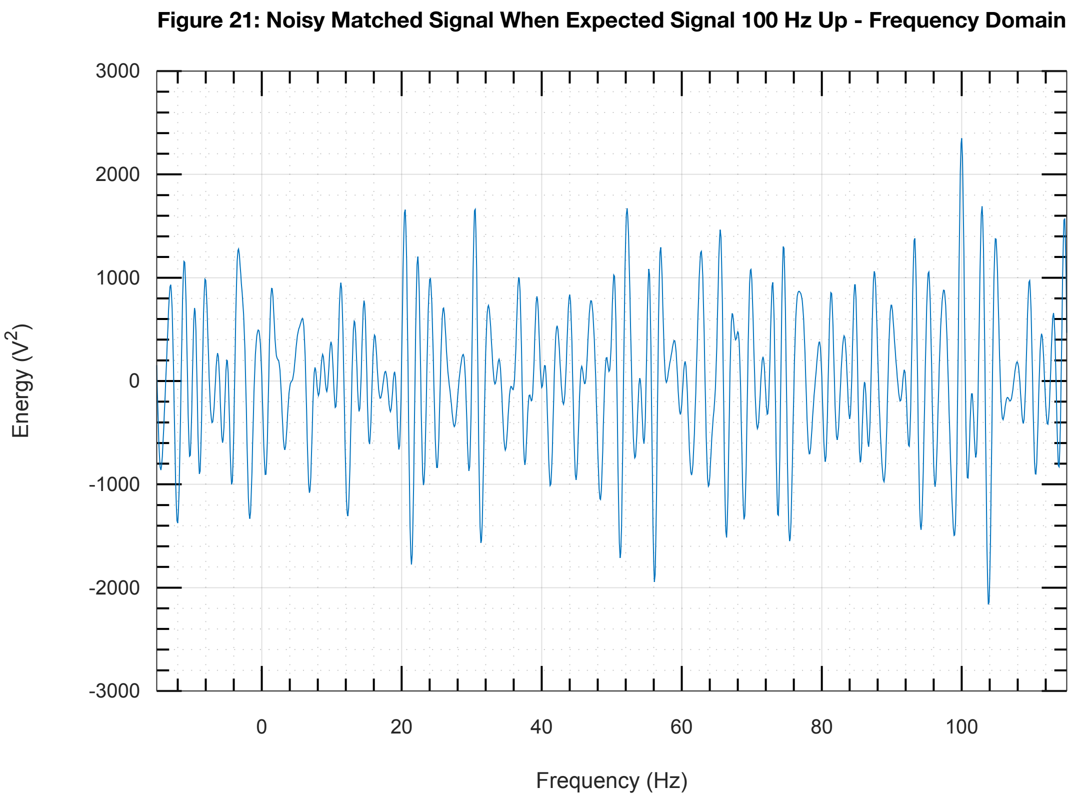
The shift of 100 Hz can be seen clearly in the plot.

The performance will be examined in the face of noise using an SNR of -26 dB again (Figure 20 & Figure 21).

```
In [24]: snr = -26; # dB
         V_matched_noisy = V_matching .* add_awgn_noise(V_new_signal, snr_dB);

         plot(T_base, V_matched_noisy, T_base, V_matched)
         ylabel("Energy (V^2)")
         xlabel("Time (s)")
         title("Figure 20: Noisy Matched Signal When Expected Signal 100 Hz Up - Tim
         grid on
         grid minor
         axis([-0.5 1.5 ], "tic")
```



In [25]:
```
E_matched = fftshift(fft(V_matched_noisy, sample_rate*10 + 1));
F_base = linspace(-sample_rate/2, sample_rate/2, size(E_matched,2));

plot(F_base, E_matched)
ylabel("Energy (V^2)")
xlabel("Frequency (Hz)")
title("Figure 21: Noisy Matched Signal When Expected Signal 100 Hz Up - Fre
grid on
grid minor
axis([-15 115], "tic")
```



It can be seen that the offset of the expected signal can read off the frequency graph even when the expected signal is swamped with noise.

Summary

In this notebook I have:

- Introduced the matched filter, how it works in the time domain, and shown how well it works in the face of noise,
- Presented a brief refresher on how frequency domain analysis of a time domain signal can be done using the fast fourier transform (FFT),
- Shown how the matched filter in the *frequency* domain can be used to determine the frequency offset of an expected signal.

In an upcoming notebook I'll determine what makes a good expected signal to be used for the matching filter and how it can be generated.