Ge193 Imaging radar and application

Homework 5: Range migration

In [ ]:

```python
## Import modules

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from tqdm import tqdm

matplotlib.rcParams.update({'font.size': 16})
```

In [ ]:

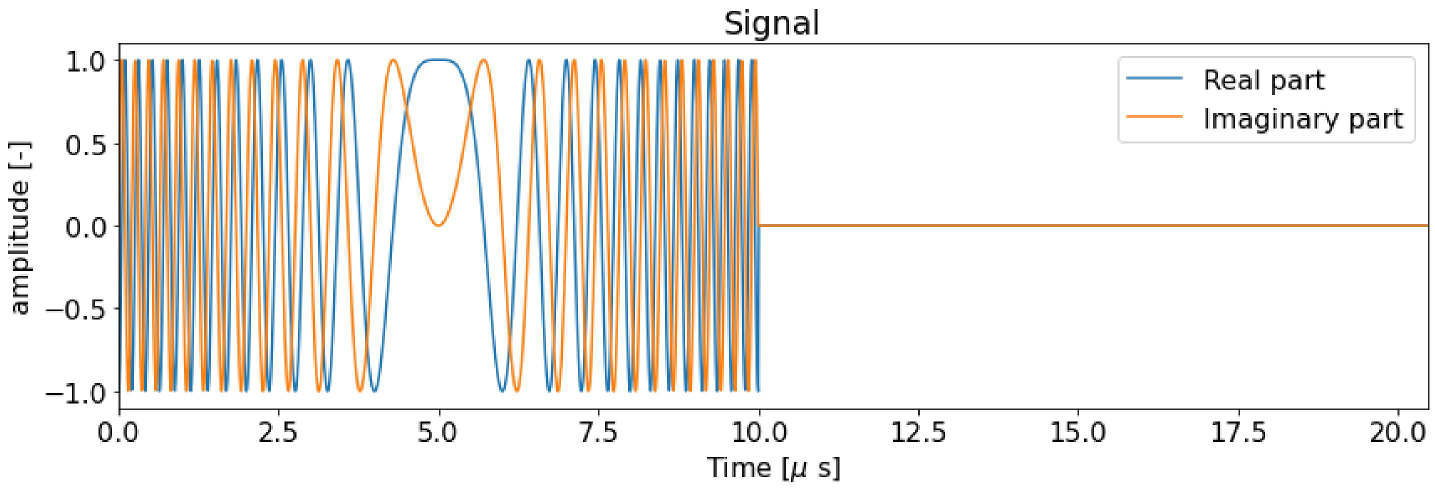
```python
## Define functions

def readuint8(file, nsamp, nlines):
    with open(file, 'rb') as fn:
        load_arr = np.frombuffer(fn.read(), dtype=np.int8)
        load_arr = load_arr.reshape((nlines, nsamp))
    return np.array(load_arr)


def readcsingle(file, nsamp, nlines):
    with open(file, 'rb') as fn:
        load_arr = np.frombuffer(fn.read(), dtype=np.csingle)
        load_arr = load_arr.reshape((nlines, nsamp))
    return np.array(load_arr)


def plot_img(data, nhdr=0, title='Data', scale=1, cmap='gray', vlim=[None,None], origin='upper', aspe
    if scale > 1:
        clabel = 'Value * {} [-]'.format(scale)
    else:
        clabel = 'Value [-]'

    # Adjust the data part for better visualization
    val = np.array(data)
    val[:,nhdr:] = scale * val[:,nhdr:]

    # plot the 2D image
    plt.figure(figsize=figsize)
    im   = plt.imshow(val, cmap=cmap, interpolation=interpolation, vmin=vlim[0], vmax=vlim[1], origin
    cbar = plt.colorbar(im, shrink=0.3, pad=0.02)
    cbar.set_label(clabel, rotation=270, labelpad=30)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xlim(lim[0], lim[1])
    plt.ylim(lim[2], lim[3])
    if (yticks is not None) and (len(yticks)==3):
        plt.yticks(np.linspace(lim[2], lim[3], yticks[2]), np.linspace(yticks[0], yticks[1], yticks[2
    if savetif is not None:
        plt.savefig('{}'.format(savetif), format = 'tif')
    plt.show()


def makechirp(N, slope, tau, fs, fc=0, start=0, phi0=0):
    """ Make a reference chirp pulse
    N:     Num of points of the whole pulse    [#]
    slope: slope of the chirp                  [Hz/s]
    tau:   chirp length                        [s]
    fs:    sample rate                         [Hz]
    fc:    central carrier freq                [Hz]
    start: starting sample # of the chirp     [#]
    """
    dt    = 1/fs                                      # sampling time interval          [s]
    npts  = tau * fs                                  # num of points of the pure chirp [#]
    t     = dt * np.arange(-npts/2, npts/2)           # time axis of the pure chirp     [s]
    phase = np.pi*slope*(t**2) + 2*np.pi*fc*t + phi0  # chirp phase                     [rad]
    chirp = np.exp(1j*phase)                          # chirp                           (cmplx)
    chirp = np.pad(chirp, (start,N-len(chirp)-start)) # pad zeros at tail and beginning (cmplx)
    t_arr = dt * np.arange(0, N)                      # time axis of the whole pulse    [s]
    #print('Chirp starts from {} samples, {} mu s'.format(start, 1e6*(start/fs)))
    return chirp, t_arr


def matched_filter(sig, ref):
    sig_fft = np.fft.fft(sig)                         # transform the signal to freq domain
    ref_fft = np.fft.fft(ref)                         # transform the reference chirp to freq domai
    spec    = sig_fft * np.conjugate(ref_fft)         # cross-correlation gives the spectrum
    comp    = np.fft.ifft(spec)                       # inverse transform it back to time domain
    return comp, spec


def plot_freq(freq, val, title, x='Frequency', y='20*log10(|spectrum|), [dB]', xlim=[None,None], ylim
    x += ' [{}]'.format(unit)
    if unit == 'MHz':
        u = 1e-6
    elif unit == 'Hz':
        u = 1
    if shift:
        val = np.fft.fftshift(val)
    plt.figure(figsize=[14,4])
    plt.plot(freq*u, val)
    plt.title(title)
    plt.xlim(min(freq)*u, max(freq)*u)
    plt.xlim(xlim[0], xlim[1])
    plt.ylim(ylim[0], ylim[1])
    plt.xlabel(x)
    plt.ylabel(y)
    plt.show()


def plot_time(t, val, title, x=r'Time', y='amplitude [-]', xlim=[None,None], ylim=[None,None], unit='
    if unit == 'micros':
        x += r' [$\mu$ s]'
        u = 1e6
    elif unit == 's':
        x += r' [s]'
        u = 1
    if shift:
        val = np.fft.fftshift(val)
    plt.figure(figsize=[14,4])
    if plotcomplex:
        plt.plot(t*u, np.real(val), label='Real part')
        plt.plot(t*u, np.imag(val), label='Imaginary part')
        plt.legend(loc='upper right')
    else:
        plt.plot(t*u, val)
    plt.title(title)
    plt.xlim(min(t)*u, max(t)*u)
    plt.xlim(xlim[0], xlim[1])
    plt.ylim(ylim[0], ylim[1])
    plt.xlabel(x)
    plt.ylabel(y)
    plt.show()


def magdB(val):
    mag = 20 * np.log10(np.abs(val) + 1e-30)
    return mag
```

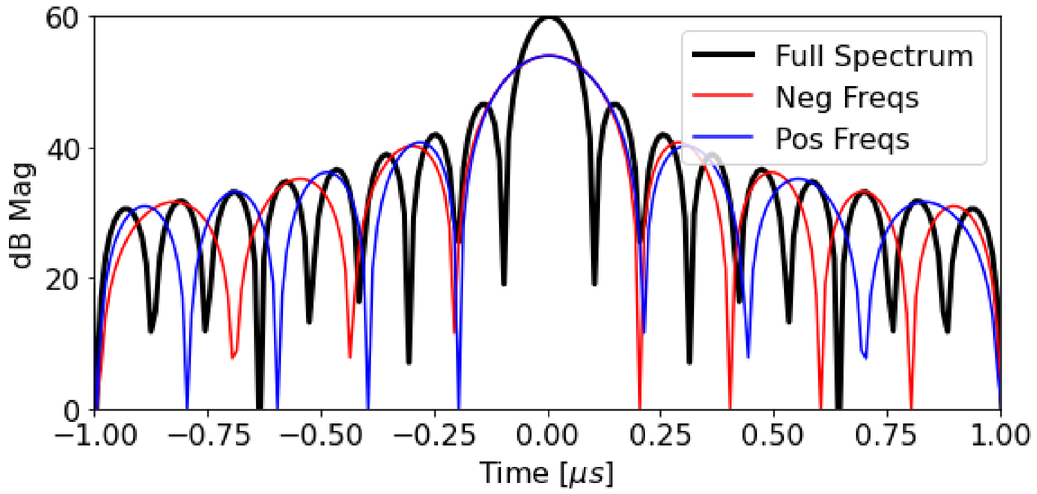
```python
def doppler_phase_shift(R, PRF):
    Naz, Nr = R.shape
    psRg = np.zeros(Nr, dtype=np.complex128)
    for i in np.arange(2,Naz):
        psRg += R[i,:] * np.conjugate(R[i-1,:])
    # compute phase shift at each range bin
    phi = np.arctan(np.imag(psRg)/np.real(psRg))
    phi_avg = np.mean(phi)
    fd = PRF * phi_avg/(2*np.pi)
    return fd, phi_avg

def doppler_cent_squint(fd, v, theta, wavelength):
    sine_squint = fd * wavelength * 0.5 * v / np.sin(theta)
    squint = np.arcsin(sine_squint)
    return squint


def roll_and_pad(x, shift):
    y = np.roll(x,-shift)
    if shift <= 0:
        y[:-shift] = 0
    elif shift > 0:
        y[-shift:] = 0
    return y
```

In [ ]:

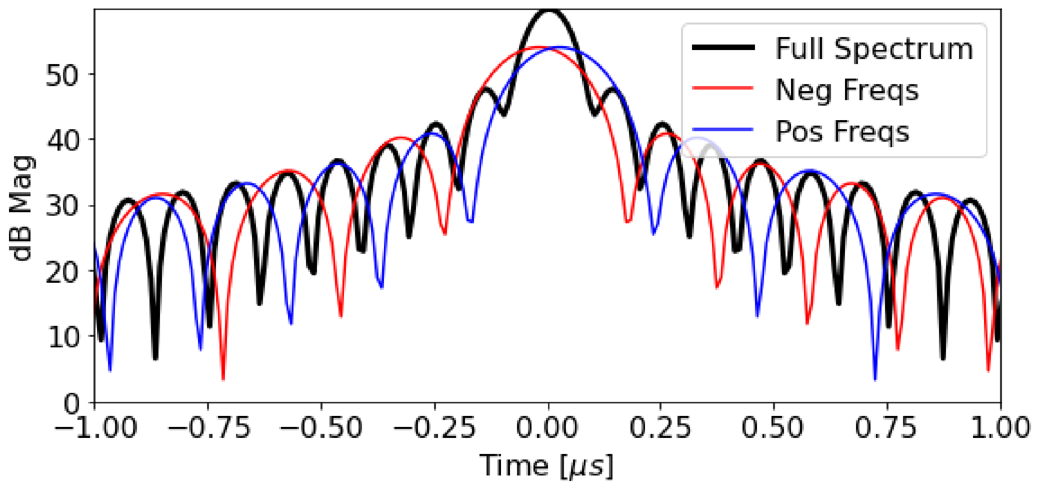
```python
# 1. Write an autofocus program to implement the sub-aperture shift algorithm.

# Use a 1-D image for ease of implementation. Assume the signal is a single chirp waveform with the f
# Chirp slope:      10^12   Hz/s
# Pulse length:     10 mu   s
# Sample rate fs:   100     MHz

## Signal
N     = 2048        # choose arbitrarily
slope = 1e12        # Hz/s
tau   = 10e-6       # sec
fs    = 100e6       # Hz
BW    = slope * tau # bandwidth [Hz]
print('One part of the time-bandwidth product = {}'.format(tau*BW/100))

## References
slope1 = 1e12
slope2 = 1.01e12
slope3 = 1.03e12
slope4 = 0.98e12

sig, t_arr = makechirp(N, slope, tau, fs)
plot_time(t_arr, sig, title='Signal', unit='micros', plotcomplex=True)

for slope_i in [slope1, slope2, slope3, slope4]:
    bw = slope_i * tau
    ref0, t_arr = makechirp(N, slope_i, tau  , fs, fc=0)                                # who
    ref1, t_arr = makechirp(N, slope_i, tau/2, fs, fc=0-bw/4)                           # low
    ref2, t_arr = makechirp(N, slope_i, tau/2, fs, fc=0+bw/4, start=int(np.round(tau/2*fs)))    # upp

    comp0, spec0 = matched_filter(sig, ref0)
    comp1, spec1 = matched_filter(sig, ref1)
    comp2, spec2 = matched_filter(sig, ref2)

    t = t_arr - np.mean(t_arr)

    plt.figure(figsize=[9,4])
    plt.plot(t*1e6, magdB(np.fft.fftshift(comp0)), c='k', label='Full Spectrum', lw=3)
    plt.plot(t*1e6, magdB(np.fft.fftshift(comp1)), c='r', label='Neg Freqs')
    plt.plot(t*1e6, magdB(np.fft.fftshift(comp2)), c='b', label='Pos Freqs')
    plt.xlim([-1, 1])
    plt.ylim([0,np.max(magdB(np.fft.fftshift(comp0)))])
    plt.ylabel('dB Mag')
    plt.xlabel(r'Time [$\mu s$]')
    plt.legend(loc='upper right')
    plt.show()

    if False: # Simply find the peak index
        peak1 = np.argmax(magdB(np.fft.fftshift(comp1)))
        peak2 = np.argmax(magdB(np.fft.fftshift(comp2)))
        delta_p = peak1 - peak2
        delta_t = delta_p / fs    # also = t[peak1] - t[peak2]
    if True: # Do cross-correlation to find the offset
        corr = np.fft.fftshift(np.fft.ifft(np.fft.fft(np.abs(comp1)) * np.conjugate(np.fft.fft(np.abs
        delta_p = np.argmax(np.abs(corr)) - N/2
        delta_t = delta_p / fs

    print('Offset = {} pixels'.format(delta_p))
    print('Delta t = {:.3e} sec'.format(delta_t))

    delta_s = 2 * slope_i**2 * delta_t / bw
    print('Slope error = {:.3e} Hz/s'.format(delta_s))
```

```
One part of the time-bandwidth product = 1.0000000000000002
```

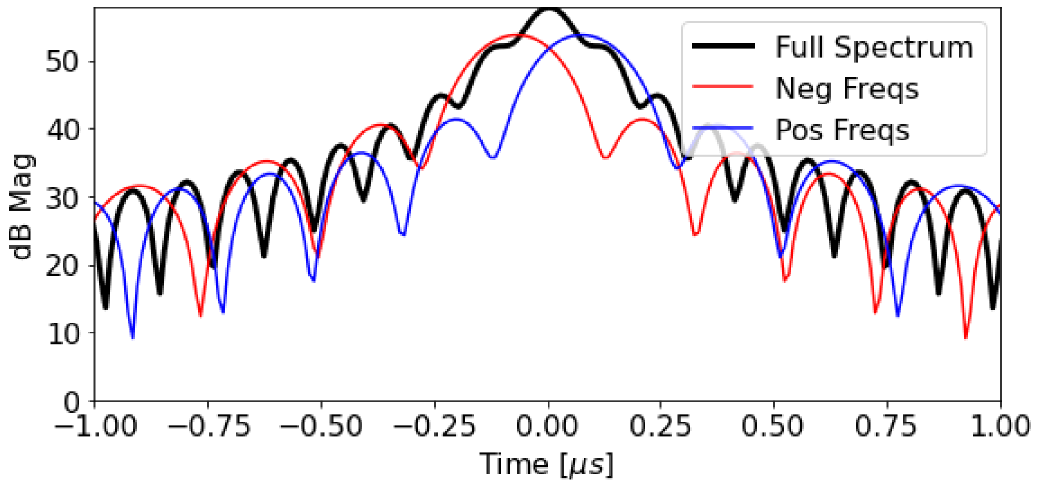




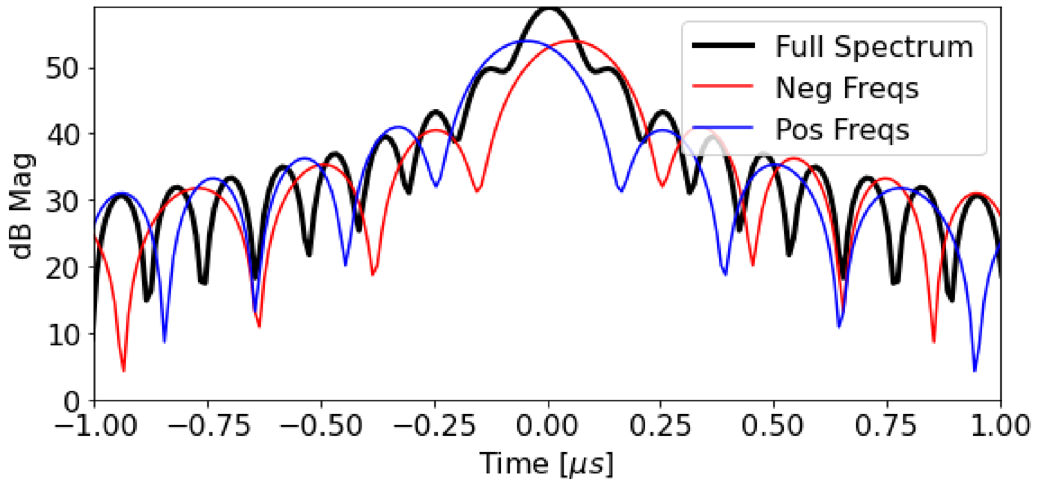
```
Offset = -1.0 pixels
Delta t = -1.000e-08 sec
Slope error = -2.000e+09 Hz/s
```



```
Offset = -5.0 pixels
Delta t = -5.000e-08 sec
Slope error = -1.010e+10 Hz/s
```



```
Offset = -15.0 pixels
Delta t = -1.500e-07 sec
Slope error = -3.090e+10 Hz/s
```



```
Offset = 10.0 pixels
Delta t = 1.000e-07 sec
Slope error = 1.960e+10 Hz/s
```

For part (a), we compress the original signal, a chirp with slope of s = 1012 Hz/s, by correlating it with the reference chirp twice. There are two subaperture reference chirps with slope s'. The first (upper-band) has positive frequencies, half of the original bandwidth. The second (lower-band) with only the negative frequencies (i.e. the other half of the bandwidth), and also half of the original bandwidth.

The compressed signals from the two subapertures will be slightly offset from each other (except for the case with identical slopes, $s' = s$), and this offset will determine Δt. We can find the offset by cross-correlating the amplitudes of the compressed signals to find the pixel offset Δp.

For part (b) Once we have the shift in pixels, we can convert to the time shift Δt by dividing by the sample rate f_s:

$$\Delta t = \frac{\Delta p}{f_s}$$

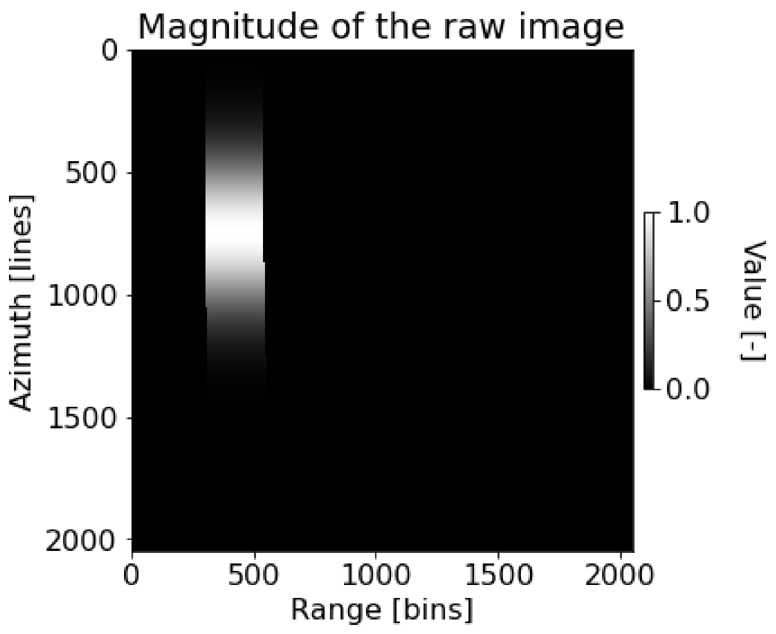
Then, with the time shift we can compute the chirp slope correction Δs, which should match the expected slope correction (which we know because we know the true slope).

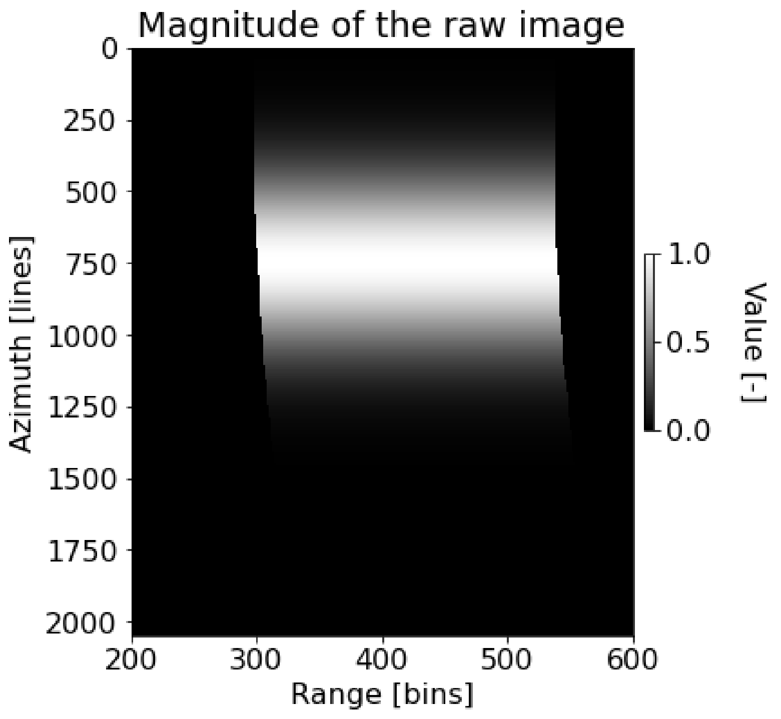
$$\Delta s = \frac{2s^2\Delta t}{BW} = \frac{2s^2\Delta t}{s\tau} = \frac{2s\Delta t}{\tau}$$

I get the following. After rounding the Δs to the second decimal point, the result is consistent to the ground truth:

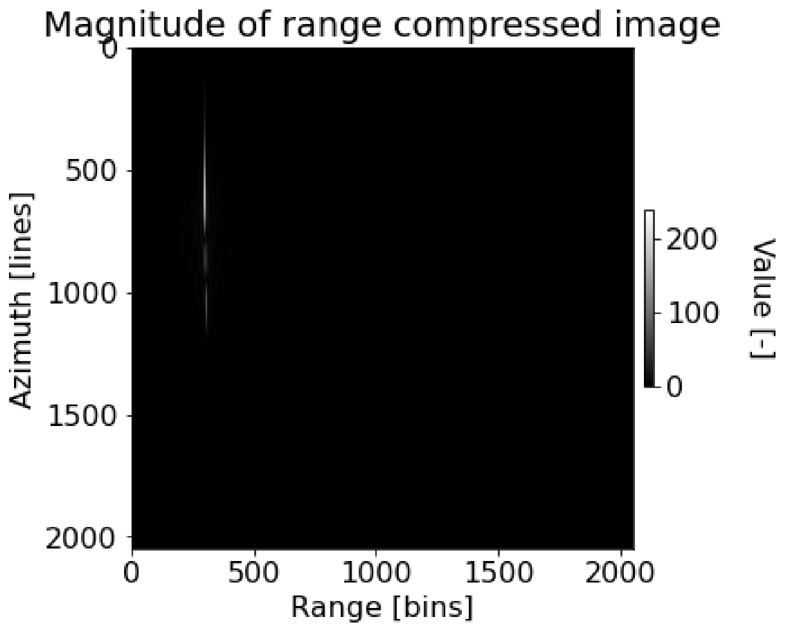
s' **[Hz/s]**	Δp **[pixels]**	Δt **[s]**	Δs **[Hz/s]**
1.00×10^{12}	~0	~0	~0
1.01×10^{12}	-5	-5×10^{-8}	-0.01×10^{12}
1.03×10^{12}	-15	-15×10^{-8}	-0.03×10^{12}
0.98×10^{12}	10	10×10^{-8}	$+0.02 \times 10^{12}$

In [ ]:
```
## Read the file and display

nsamp = 2048     # num of complex samples (columns)
nline = 2048     # num of lines (rows)

sig = readcsingle('simlband.dat', nline, nsamp)

plot_img(np.abs(sig), title='Magnitude of the raw image')
plot_img(np.abs(sig), title='Magnitude of the raw image', lim=[200,600,None,None], aspect='auto')
```

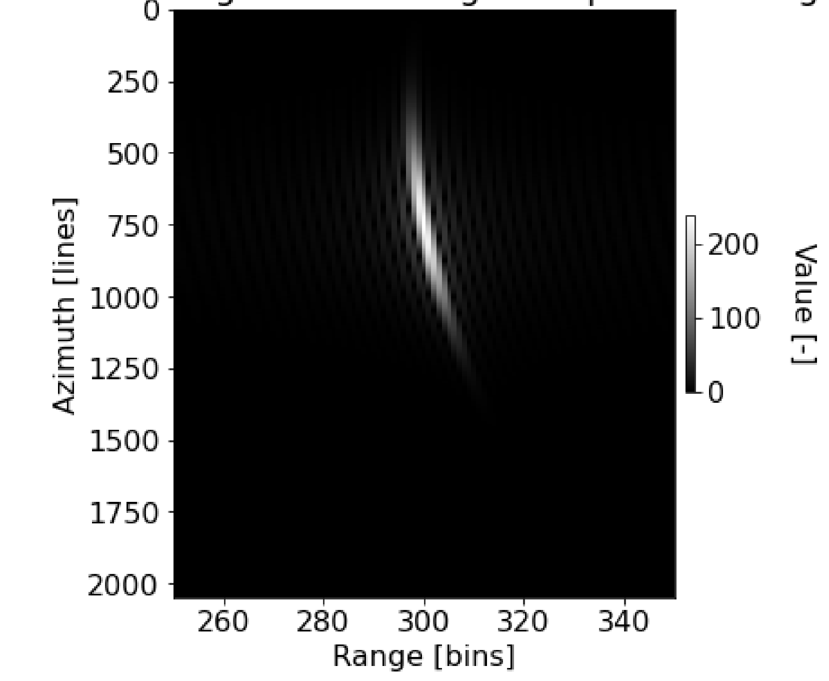




```
In [ ]: ## Range compress and display

        slope = 1e12     # Hz/s
        tau   = 10e-6    # s
        fs    = 24e6     # Hz
        ref, t_arr = makechirp(nsamp, slope, tau, fs)

        sig_comp, _  = matched_filter(sig, ref)

        plot_img(np.abs(sig_comp), title='Magnitude of range compressed image')
        plot_img(np.abs(sig_comp), title='Zoom-in magnitude of range compressed image', lim=[250,350,None,Non
```



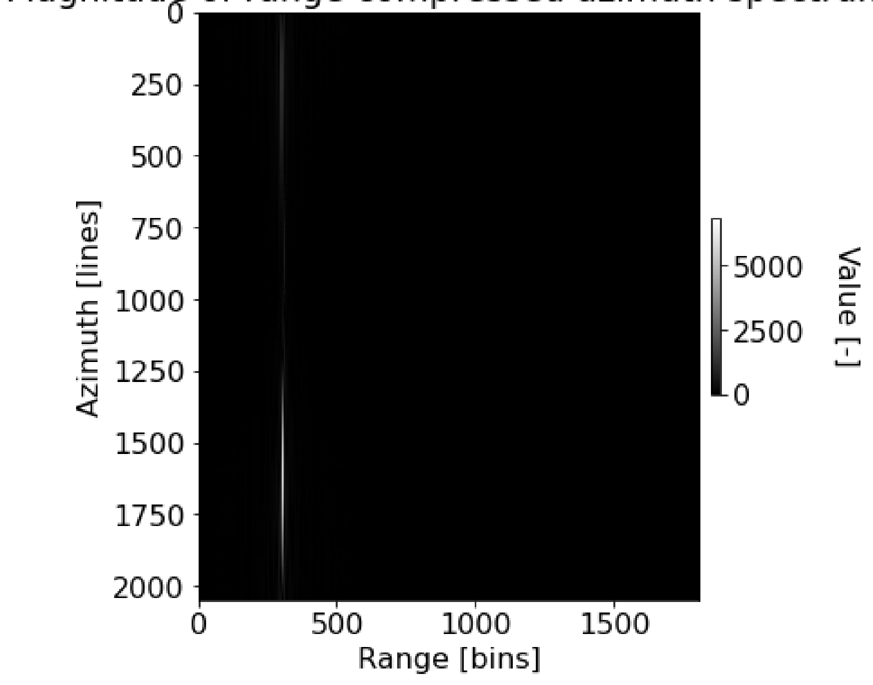
Zoom-in magnitude of range compressed image

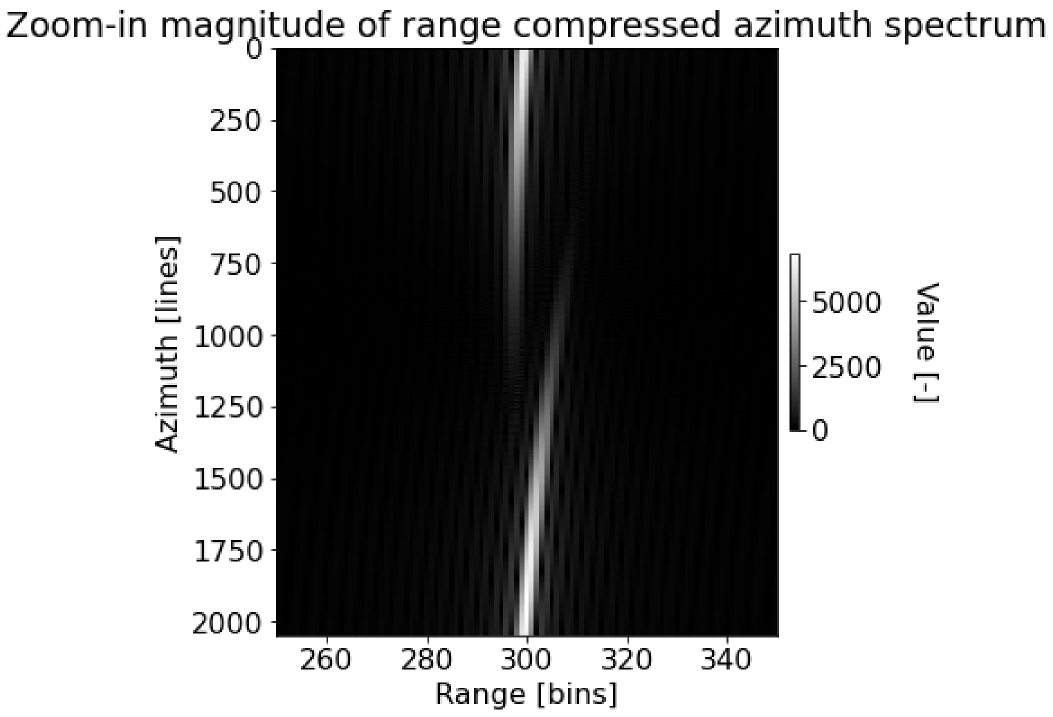


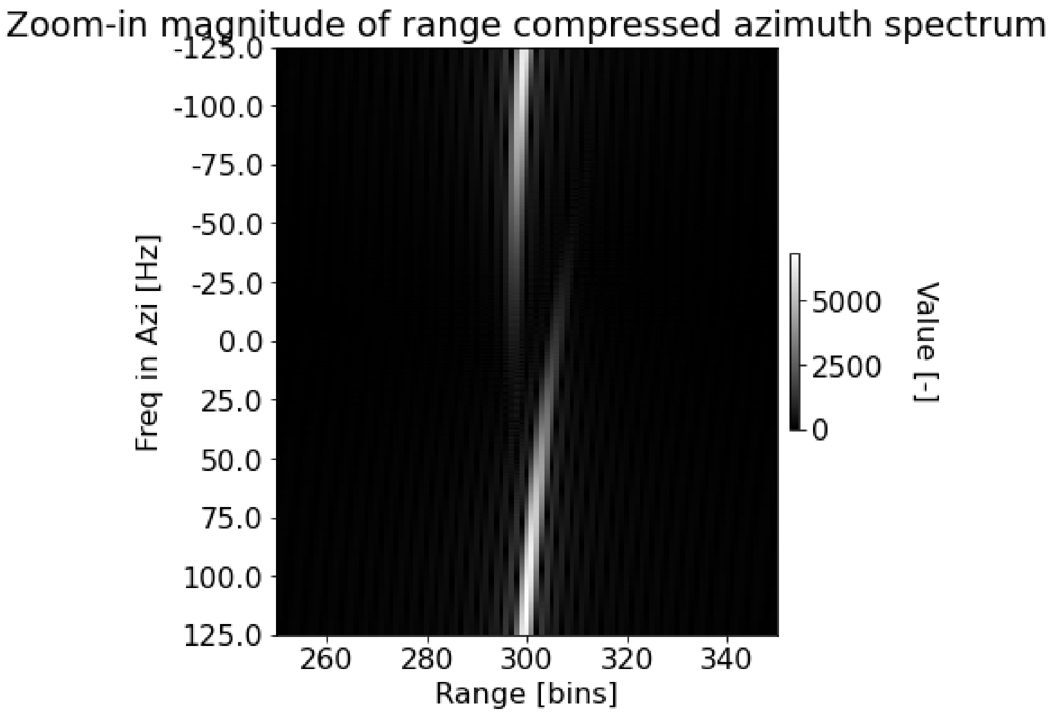
In [ ]:

```
## Azimuth transform and display
PRF  = 250       # Hz
vx   = 250       # m/s
l    = 2         # m
wl   = 0.25      # m
r0   = 4653      # m

nvalid = int(nsamp - tau*fs)

sig_comp_azspec = np.fft.fft(sig_comp[:,:nvalid], axis=0)

sig_comp_azspec_shift = np.fft.fftshift(np.fft.fft(sig_comp[:,:nvalid], axis=0), axes=0)

plot_img(np.abs(sig_comp_azspec_shift), title='Magnitude of range compressed azimuth spectrum', aspec
plot_img(np.abs(sig_comp_azspec_shift), title='Zoom-in magnitude of range compressed azimuth spectrum
#plot_img(np.abs(sig_comp_azspec_shift), title='Magnitude of range compressed azimuth spectrum', ylab
plot_img(np.abs(sig_comp_azspec_shift), title='Zoom-in magnitude of range compressed azimuth spectrum
```

Magnitude of range compressed azimuth spectrum



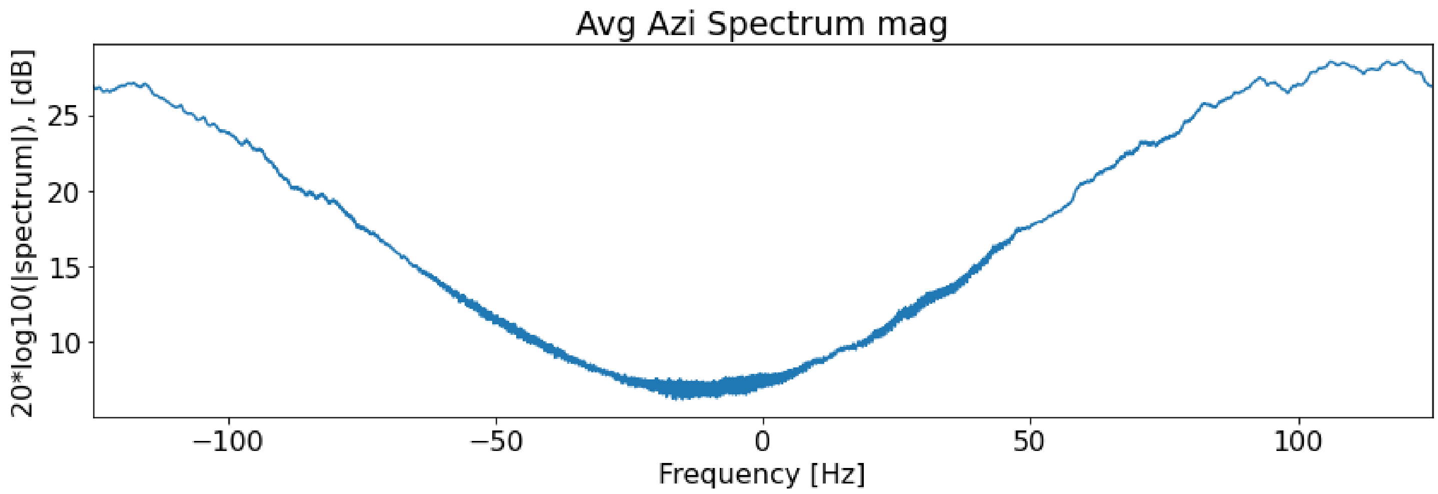




After transforming the range compressed data in the azimuth direction, we still see range migration between the two tail ends of the spectra (about 15-20 pixels), especially when we zoom in on range bins 250-350. Notice that f_{DC} is around $\frac{PRF}{2}$, which explains why the "ends" of the spectra come together around -10 Hz. Also, the azimuth frequency range can have an ambiguity of n*prf, so we only plot the axis label as [$\frac{-PRF}{2}$, $\frac{PRF}{2}$].

In [ ]:
```
## Estimate the Doppler Centroid

sig_comp_azspec_avg = np.mean(np.abs(sig_comp_azspec_shift), axis=1)

freq = np.linspace(-PRF/2, PRF/2, nline)

plot_freq(freq, sig_comp_azspec_avg, title='Avg Azi Spectrum mag', x='Frequency', y='20*log10(|spectr

fdc = freq[np.argmax(sig_comp_azspec_avg)]

fdcs = np.arange(fdc-3*PRF, fdc+3*PRF, PRF)
```

```
print('F_DC estimated from the spectrum peak is {:.2f}'.format(fdc))
print('F_DC ambiguity: ', fdcs)
```

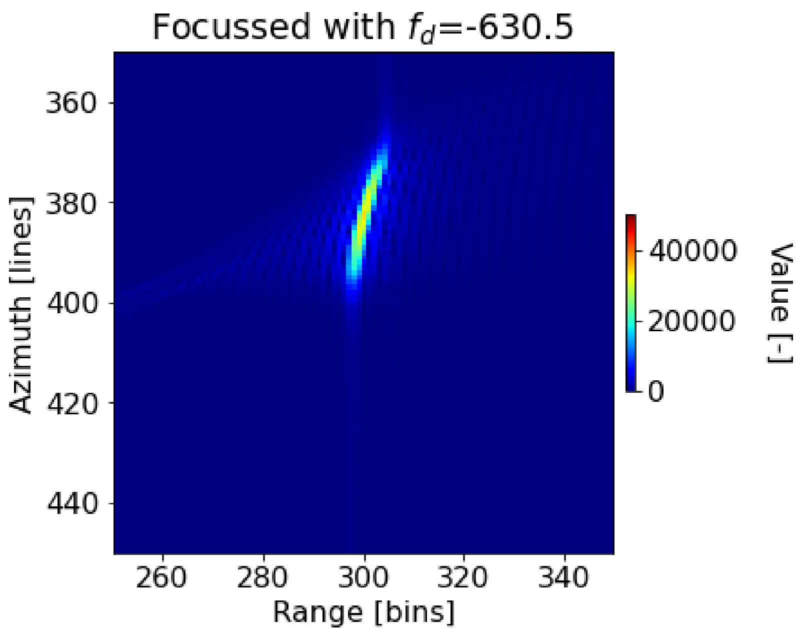


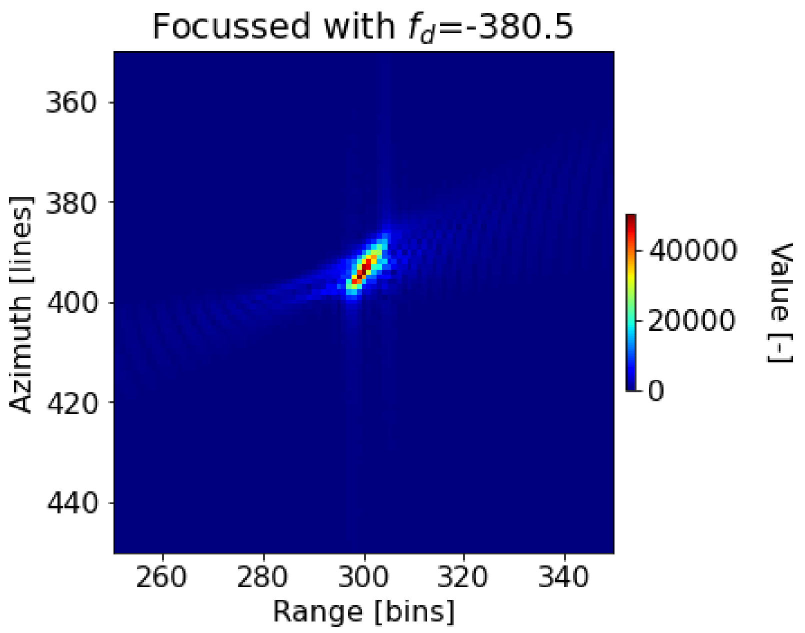
```
F_DC estimated from the spectrum peak is 119.50
F_DC ambiguity:  [-630.49584758 -380.49584758 -130.49584758  119.50415242  369.50415242
  619.50415242]
```

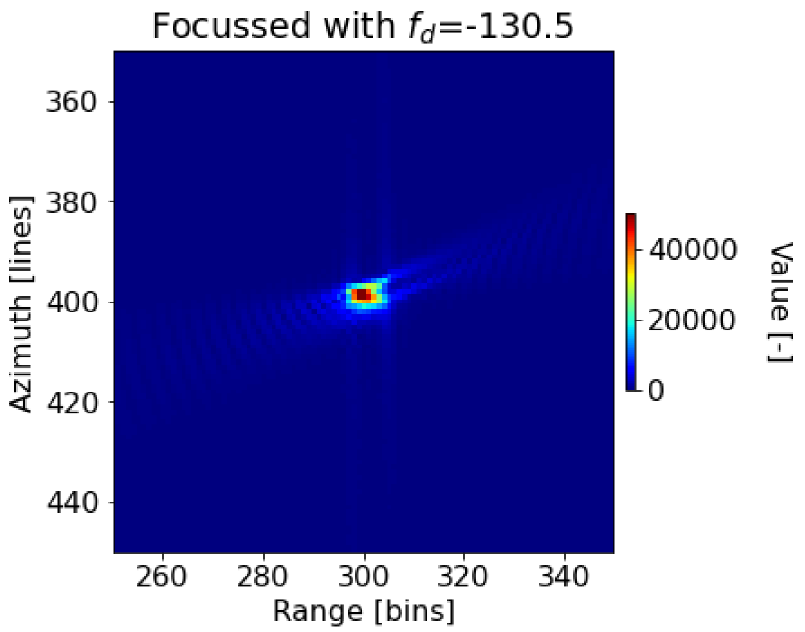
Since we really only have one scatterer, the average phase shift method couldn't be effectively used. Therefore, I estimated the Doppler centroid by finding the peak of the average spectrum and decided on Fdc = 119.5 Hz. To account for the ambiguity, I also added and subtracted multiples of the PRF=250 Hz. Some of the possible f_{dc} values are: -630.5, -380.5, -130.5, 119.5, 369.5, 619.5.

In [ ]:

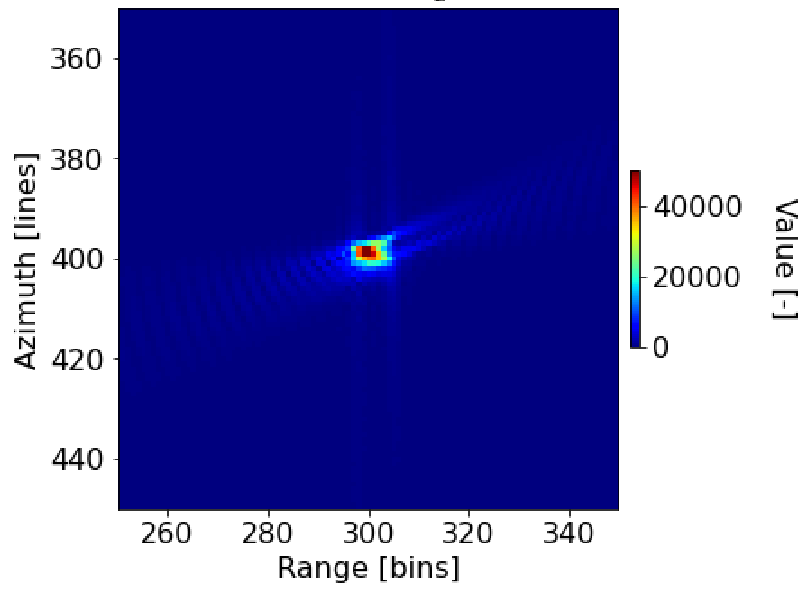
```
## Focus processing

c  = 3e8
BW = tau * slope
dr = c/(2*BW)     # slant ragne resolution
drbin = c/(2*fs) # slant range bin spacing


# Try different Doppler centroids
for k in np.arange(len(fdcs)):
    fd = fdcs[k]

    # Azimuth focus processing
    focused = np.zeros([nline, nvalid], dtype=np.complex128)
    for i in np.arange(nvalid):
        ri        = r0 + (i * drbin)
        xi        = fd * wl * ri / 2 / vx        # assuming constant Doppler centroid
        r_dc_i    = np.sqrt(ri**2 + xi**2)
        fr_i      = -2*(vx**2)/r_dc_i/wl
        tau_az_i = 0.8 * r_dc_i * wl / vx / l

        # create azimuth reference chirp
        ref_az    = makechirp(N=nline, slope=fr_i, tau=tau_az_i, fs=PRF, fc=fd)[0]

        # process the signal in one patch (all 2048 lines)
        signal         = sig_comp[:, i]
        #focused[:, i] = matched_filter(signal, ref_az)[0]
        focused[:, i] = np.fft.ifft(sig_comp_azspec[:, i] * np.conjugate(np.fft.fft(ref_az)))


    # Plot the focussed image (zoom-in)
    plot_img(np.abs(focused), title=r'Focussed with $f_d$={:.1f}'.format(fd), cmap='jet', lim=[250,35
```


Focussed with f_d=-630.5
Azimuth [lines]
Range [bins]
Value [-]
0
20000
40000
360
380
400
420
440
260
280
300
320
340

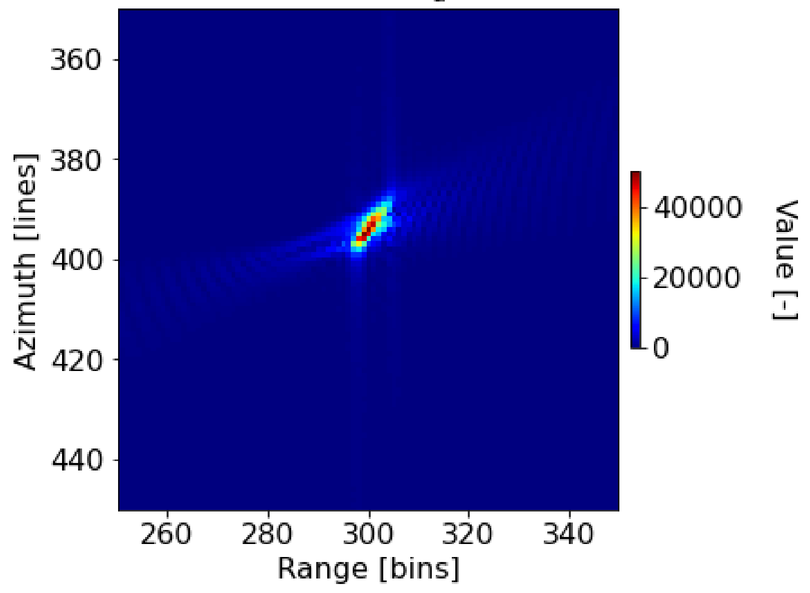

Focussed with f_d=-380.5
Azimuth [lines]
Range [bins]
Value [-]
0
20000
40000
360
380
400
420
440
260
280
300
320
340


Focussed with f_d=-130.5
Azimuth [lines]
Range [bins]
Value [-]
0
20000
40000
360
380
400
420
440
260
280
300
320
340

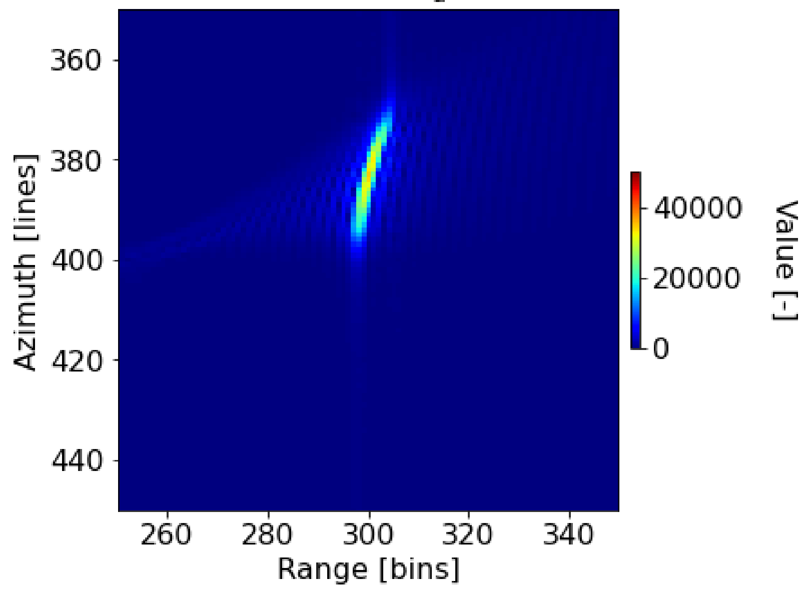
Focussed with f_d=119.5



Focussed with f_d=369.5



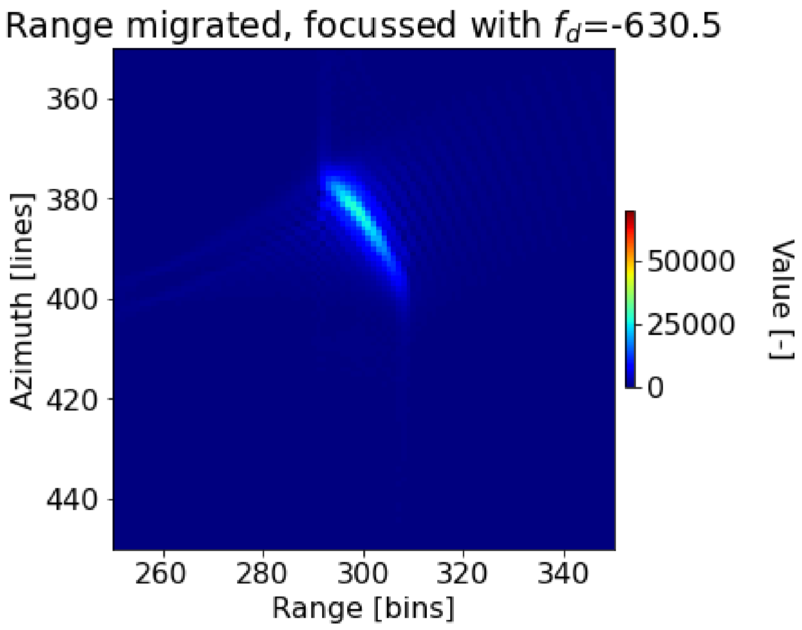
Focussed with f_d=619.5



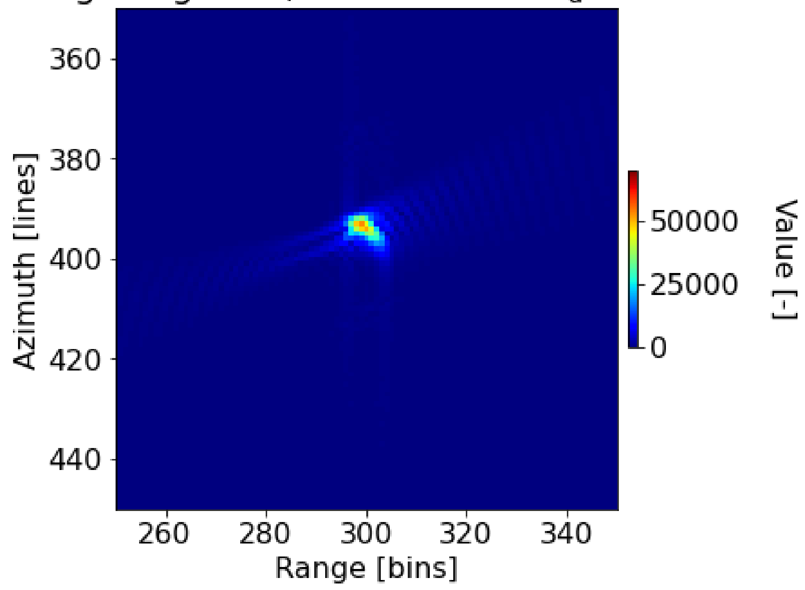
Although the point looks pretty good when zoomed out far, when we zoom in we can see some significant blurring, or spreading out of the signal over range bins. The best response is when we use a doppler centroid of -130.5 Hz or 119.5

Hz, although our original pick of 108 Hz does almost just as well. 619.5 Hz does the worst, since it has the most smearing.

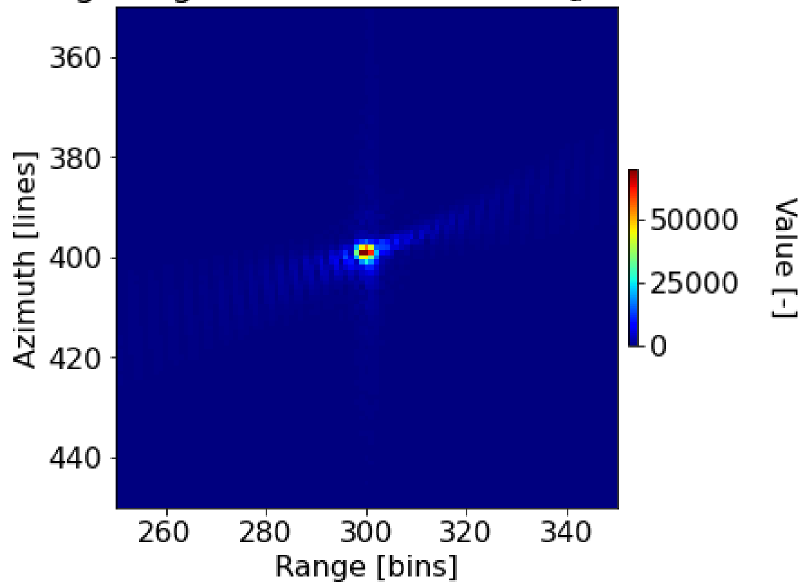
```
In [ ]:
## Apply "Cut and Paste" algorithm for range migration. Assume each f_dc found above. Whis gives the


# Different Doppler centroids
for k in np.arange(len(fdcs)):
    fd = fdcs[k]

    # pre-allocate the array
    focused_fft = np.zeros([nline, nvalid], dtype=np.complex128)     # freq domain to catch the migra
    focused_rm  = np.zeros([nline, nvalid], dtype=np.complex128)     # time domain to catch the azi m

    # apply cut-and-paste method (with linear interpolation)
    for j in np.arange(nline):
        f      = PRF * j/nline
        nn = np.rint((f-fd)/PRF)
        f     -= nn * PRF
        offset = (f**2-fd**2) * wl**2 * r0 / (8*vx**2)  # find the range difference wrt rdc (non-desk
        shift  = offset/drbin                           # shift = range history / range bin spacing
        shift1 = int(shift)                             # lowest integer shift
        shift2 = shift1 + int(np.sign(shift))           # highest integer shift
        w1  = np.abs(shift2 - shift)                    # weight 1
        w2  = np.abs(shift - shift1)                    # weight 2 (set w2=0, and shift1=nearest inte
        # linear interpolation between the nearby two pixels
        focused_fft[j, :] = w1 * roll_and_pad(sig_comp_azspec[j, :], shift1) + w2 * roll_and_pad(sig_


    # Azimuth focus processing
    for i in np.arange(nvalid):
        ri       = r0 + (i * drbin)
        xi       = fd * wl * ri / 2 / vx        # assuming constant Doppler centroid
        r_dc_i   = np.sqrt(ri**2 + xi**2)
        fr_i     = -2*(vx**2)/r_dc_i/wl
        tau_az_i = 0.8 * r_dc_i * wl / vx / l

        # create azimuth reference chirp
        ref_az   = makechirp(N=nline, slope=fr_i, tau=tau_az_i, fs=PRF, fc=fd)[0]

        # process the signal in one patch (all 2048 lines)
        focused_rm[:, i] = np.fft.ifft(focused_fft[:, i] * np.conjugate(np.fft.fft(ref_az)))


    # Plot the focussed image (zoom-in)
    plot_img(np.abs(focused_rm), title=r'Range migrated, focussed with $f_d$={:.1f}'.format(fd), cmap
```



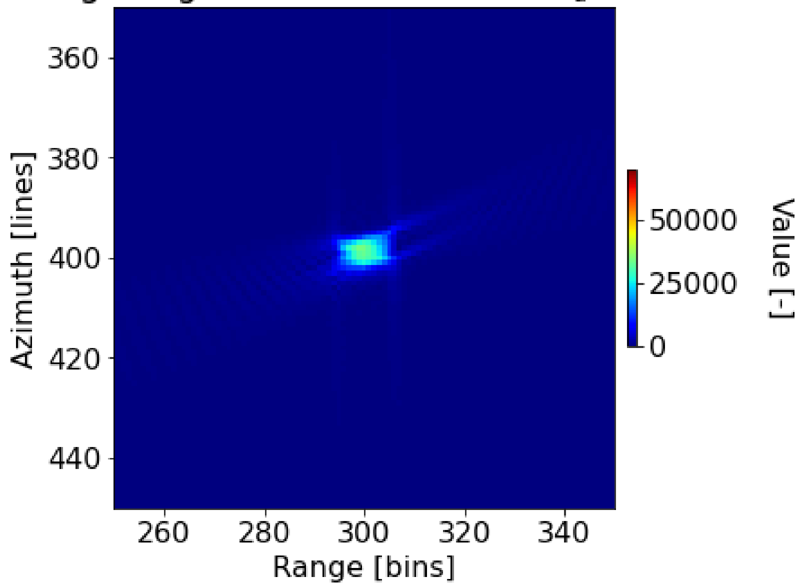
Range migrated, focussed with f_d=-380.5

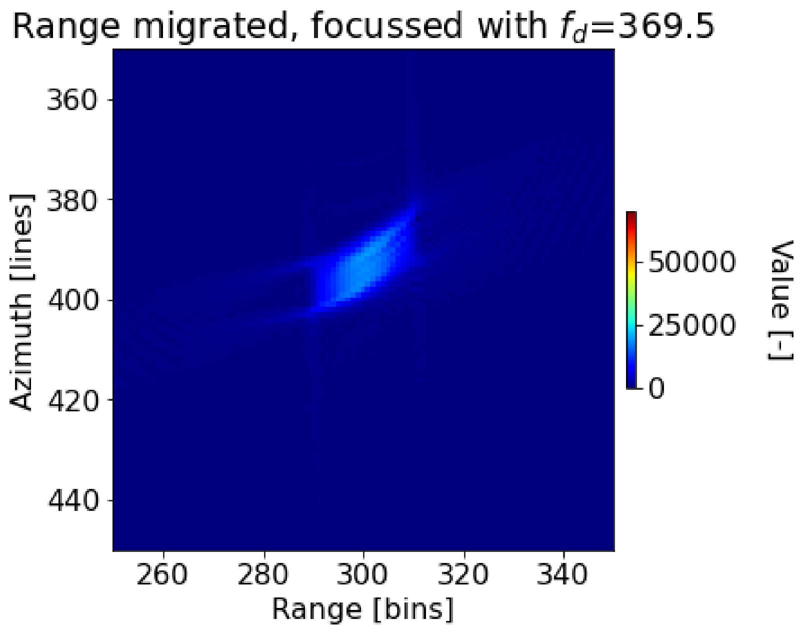


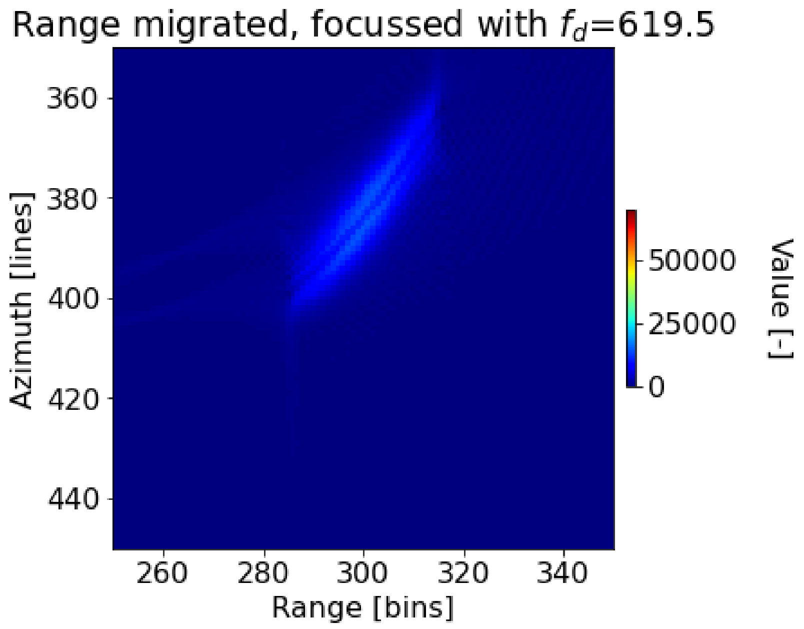
Range migrated, focussed with f_d=-130.5



Range migrated, focussed with f_d=119.5







Using the cut and paste algorithm (with simple linear interpolation), we get the best concentration of the returned signal with a doppler centroid of -130.5 Hz. The others have a significant smearing of the returned energy. However, note that our corrected returned signal is still not perfect. We would likely see better results if we also added more complex interpolation to our cut-and-paste algorithm. Also note that the range migration process increased the sensitivity to the Doppler Centroid ambiguity.

```
In [ ]:
```