Experiments with the MUSIC Algorithm

Updated March 28th, 2023

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import find_peaks
from tqdm import tqdm
```

Support Functions

In [ ]:
```
#enabling random configurations
freq_range = [2,10]
min_freq_diff = 2
def generate_random_frequencies(freq_range,min_freq_diff,D_signals):
    """Randomly select D different frequencies with a minimum difference and w

    Args:
        freq_range (np.array): [min frequency, max frequency]
        min_freq_diff (float): the minimum separation between frequencies
        D_signals (int): the number of signals

    Returns:
        np.array: A list with D_signals different signal frequencies
    """

    #come up with an array of options
    a = np.arange(freq_range[0],freq_range[1],min_freq_diff)

    #randomly select from the list
    return np.random.choice(a,size=D_signals,replace=False)

def generate_random_angles(AoA_range,min_angular_diff,D_signals):
    """Randomly select D different angles of arrival with a minimum difference

    Args:
        AoA_range (np.array): [min frequency, max frequency]
        min_angular_diff (float): the minimum separation between angles
        D_signals (int): the number of signals

    Returns:
        np.array: A list with D_signals different angles for each signal
    """

    #come up with an array of options
    a = np.arange(AoA_range[0],AoA_range[1],min_angular_diff)

    #randomly select from the list
    return np.random.choice(a,size=D_signals,replace=False)

# generating example signals and noise values
def generate_sample_signal(f,t,complex=True,plot = False):
    """Generates a sinusoidal signal to be used for testing the MUSIC algorith
```

```
    Args:
        f (float): the desired frequency (in Hz) of the sinusoid
        t (np.array): the sample times at which to generate the sinusoid
        complex (bool, optional): generates a complex sinusoid on True. Defaul
        plot (bool, optional): plots the generated signal on True. Defaults to
    """
    #generate the signal
    if complex:
        x = np.exp(2j * np.pi * f * t)
    else:
        x =  np.sin(2 * np.pi * f * t)

    #plot the signal
    if plot:
        plot_signal(x,t)

    #return the generated signal
    return x

def generate_signals(freqs,t,complex=True,verbose=False):
    """Generates a series of sample signals to be used to test the MUSIC algor

    Args:
        freqs (np.array): array of the frequencies (in Hz) of the sample sinus
        t (np.array): the sample times at which to generate the sinusoid
        complex (bool, optional): generate complex sinusoids on True. Defaults
        verbose (bool, optional): Print out the final s(t) signal matrix on Tr

    Returns:
        np.array: len(freqs) x len(t) array of sample sinusoidal signals
    """

    #initialize the s_t array
    if complex:
        s_t = np.empty((len(freqs),len(t)),dtype=np.complex_)
    else:
        s_t = np.empty((len(freqs),len(t)),dtype=float)

    #generate the signals
    for i in range(0,len(freqs)):
        s_t[i,:] = generate_sample_signal(freqs[i],t,complex=complex)

    #print the generated signals if verbose is true
    if verbose:
        out_str = "s(t) = \n{}".format(s_t)
        print(out_str)

    return s_t

def plot_signal(x,t,title = "Sine Wave"):
    """generates a plot of the sample signal

    Args:
        x (np.array): x(t) values
        t (np.array): time samples
        title (str, optional): Plot title. Defaults to "Sine Wave".
    """
    #plotting for if the signal is complex
    if np.iscomplex(x).any():
        plt.plot(t, x.real)
```

```
        plt.plot(t, x.imag)
        plt.legend(['Real', 'Imaginary'])
    else:
        plt.plot(t,x)
        plt.legend(['Signal'])
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.title(title)
    plt.show()

def compute_signal_power(x,dB = False):
    """Compute the power of a given signal (used to verify that SNR is correct

    Args:
        x (np.array): the signal to compute the power of
        dB (bool, optional): on True converts the power to dB. Defaults to Fal
    """
    N = len(x)
    pow = np.power(np.linalg.norm(x[0,:]),2) * 1/N

    if dB:
        return 10 * np.log10(pow)
    else:
        return pow

def determine_signal_amplitude(snr,sigma_2 = 1):
    """Compute the required amplitude for a complex sinusoidal signal in noise

    Args:
        snr (_type_): _description_
        sigma_2 (float,optional): the noise variance. Defaults to 1
    Returns:
        (float,float): tuple with (Signal power, signal amplitude)
    """

    P_s_t = sigma_2 * (10 ** (snr/10))

    Amp_s_t = np.sqrt(P_s_t) #not multiplying by 2 because this is a complex s

    return P_s_t,Amp_s_t

def generate_noise_samples(sigma_2, cross_corr, K_Rx, N_samples,verbose = Fals
    """Generate n_t (the noise samples) with a given signal variance and poten

    Args:
        sigma_2 (float): The noise variance for each element
        cross_corr (float): The correlation between each element and the other
        K_Rx (int): The number of receive elements
        N_samples (int): The number of noise samples to generate
        verbose(bool, optional): prints the noise covariance matrix on True, D
    """

    #assume noise has a mean of zero
    mu = np.zeros((K_Rx))

    #set the cross correlation terms of the noise covariance matrix
    cov = np.ones((K_Rx,K_Rx)) * cross_corr / 2 #also dividing by 2 because it
    np.fill_diagonal(cov,sigma_2/2) #divigin sigma squared by 2 because it is

    #generate the noise samples (have to generate real and complex separately
```

```
    noise_samples = np.random.multivariate_normal(mu,cov,size=(2,N_samples)).t

    #combine real and complex into final noise sample set
    n_t = noise_samples[:,:,0] + 1j * noise_samples[:,:,1]

    if verbose:
        out_str = "Set Noise Covariance Matrix:\n{}".format(cov * 2)
        print(out_str)

        cov_estimated = np.abs(np.cov(n_t))
        out_str = "\n\n\nActual Noise Coveriance Matrix:\n{}".format(cov_estim
        print(out_str)

    return n_t

#MUSIC Specific Functions

def compute_mode_vector(theta,K_Rx,radians=False,verbose = False):
    """Computes a mode vector for a given value of theta

    Args:
        theta (float): AoA for the given mode vecotor [in degrees (assumed) or
        K_Rx (int): The number of receivers
        radians (bool, optional): Assumes theta is in radians if true. Default
        verbose (bool, optional): prints the return matrix. Defaults to False.

    Returns:
        np.array: K_Rx x 1 mode vector for the corresponding theta
    """
    #define the k indicies
    k = np.arange(1,K_Rx + 1)

    #convert to radians if theta is given in degrees
    if not radians:
        theta = np.deg2rad(theta)

    #compute the mode vector
    a_theta = np.array([np.exp(1j * np.pi * (k - 1) * np.sin(theta))]).transpo

    if verbose:
        ret_string = "A(theta) = \n{}".format(a_theta)
        print(ret_string)

    return a_theta

def generate_mode_vector_matrix(thetas,K_rx,radians=False,verbose = False):
    """Generate the A matrix (each column is a mode vector corresponding to a

    Args:
        thetas (np.array):list of AoA angles [in degrees (assumed) or radians]
        K_rx (int): Number of receivers
        radians (bool, optional): Assumes theta is in radians if true. Default
        verbose (bool, optional): prints the return matrix. Defaults to False.

    Returns:
        _type_: _description_
    """

    #initialize empty A matrix
    A = np.empty((K_rx,len(thetas)),dtype=np.complex_)
```

```
    #compute each mode vector for a given theta
    for i in range(0,len(thetas)):
        A[:,i:i+1] = compute_mode_vector(theta=thetas[i],K_Rx=K_rx,radians=rad

    #print resulting A matrix if verbose set to true
    if verbose:
        out_str = "A Matrix: \n{}".format(A)
        print(out_str)

    return A

def compute_P_MU(theta,E_N,K_rx,radians=False):
    """Compute P_mu from the MUSIC algorithm

    Args:
        theta (float): angle to use when generating the mode vector [in degree
        E_N (np.array): K_rx x D_signals matrix whose columns are the eigen ve
        K_rx (int): number of receivers
        radians (bool, optional): Assumes theta is in radians if true. Default

    Returns:
        P_MU(theta): the music spectrum for the given angle of arrival
    """

    #compute mode vector
    a_theta = compute_mode_vector(theta,K_rx,radians)

    return 1 / (a_theta.conj().transpose() @ E_N @ E_N.conj().transpose() @ a_

def MUSIC(x_t,K_rx, D_signals,log_scale = True, plot_spectrum = True,verbose =
    """ Performs the MUSIC algorithm (optional tranparent mode to see what is

    Args:
        x_t (np.array): K_rx -by- (number of samples) samples for each receive
        K_rx (int): number of receivers
        s_t (np.array,optional): D_signals -by- (number of samples) samples fo
        A (np.array,optional): K_rx -by- D_signals matrix containing the true
        D_signals (int): The number of signals for the MUSIC algorithm to assu
        log_scale (bool, optional): Plots and returns P_mus in log scale on Tr
        plot_spectrum (bool, optional): plots the MUSIC spectrum when true. De
        verbose (bool, optional): Reports back all major calculations when Tru
        transparent_mode(bool,optional): In transparent mode, the algorithm al
            can understand what the algorithm is doing

    Returns:
        (np.array,np.array,np.array): A tuple containing (P_mus,thetas,estimat
    """
    #compute Rs, A Rs A^H, and the eigen values/vectors for the signal covaria
    if transparent_mode:
        #compute signal coveriance matrix
        R_s = np.cov(s_t)

        #compute A Rs A^H
        A_Rs_AH = A @ R_s @ A.conj().transpose()

        #compute signal eigen values and vectors
        v,w = np.linalg.eigh(A_Rs_AH)

        eigen_vals_s_t = np.flip(v)
```

```
        eigen_vectors_s_t = np.flip(w,axis=1)

    #compute sensor covariance matrix
    R_x = np.cov(x_t)

    #compute sensor eigen values and vectors
    v,w = np.linalg.eigh(R_x)
    eigen_vals_x_t = np.flip(v)
    eigen_vectors_x_t = np.flip(w,axis=1)

    #obtain the eigen vectors corresponding to the null space of Rx
    E_n = eigen_vectors_x_t[:,-1 * D_signals :]

    thetas = np.arange(-90,90,0.05)
    P_mus = np.empty(len(thetas))

    for i in range(len(thetas)):
        P_mus[i] = np.real(compute_P_MU(thetas[i],E_n,K_rx,radians=False))

    #convert to log scale if desired
    if log_scale:
        P_mus = 10 * np.log10(P_mus)

    #estimate the AoA's from the spectrum
    peak_indicies,properties = find_peaks(P_mus,height=5)
    estimated_AoAs = np.sort(thetas[peak_indicies[:D_signals]])

    if plot_spectrum:
        plot_music_spectrum(thetas,P_mus,estimated_AoAs,log_scale=log_scale)

    #print out resulting fields if desired
    if verbose:

        if transparent_mode:
            #print signal covariance matrix
            out_str = "Sensor Covariance Matrix (magnitude):\n{}".format(np.ab
            print(out_str)

            #print A Rs A^H matrix
            out_str = "\n\n\nA Rs A^H:\n\n \t Covariance Matrix (magnitude):\n
            print(out_str)

            #print eigen values
            out_str = "\t Eigen Values:\n{}".format(np.absolute(eigen_vals_s_t
            print(out_str)

            #print eigen vectors
            out_str = "\t Eigen Vectors:\n{}".format(eigen_vectors_s_t)
            print(out_str)

        #print Sensor Covariance matrix
        out_str = "\n\n\nRx:\n\n \t Covariance Matrix (magnitude):\n{}".format
        print(out_str)

        #print eigen values
        out_str = "\t Eigen Values:\n{}".format(np.absolute(eigen_vals_x_t))
        print(out_str)

        #print eigen vectors
        out_str = "\t Eigen Vectors:\n{}".format(eigen_vectors_x_t)
```

```
        print(out_str)

        #print estimated AoAs
        out_str = "\n\n\nEstimated AoAs: {}".format(estimated_AoAs)
        print(out_str)

    return P_mus,thetas, estimated_AoAs

def plot_music_spectrum(thetas,P_mus,estimated_AoAs = None, log_scale = True):
    """generates a plot of the MUSIC spectrum

    Args:
        thetas (np.array): theta values that the spectrum is taken at
        P_mus (np.array): music spectrum
        log_scale (bool, optional): Assumes P_mus are in log scale on True. De
    """
    #plotting for if the signal is complex
    plt.plot(thetas,P_mus)
    plt.legend(['Music Spectrum'])
    plt.xlabel('Theta')
    if log_scale:
        plt.ylabel('Amplitude (dB)')
    else:
        plt.ylabel('Amplitude')

    #if estimated AoAs are provided, include in the title
    if estimated_AoAs is None:
        plt.title("Music Spectrum")
    else:
        plt.title("Music Spectrum (Estimated AoAs: {})".format(estimated_AoAs)
    plt.show()

def run_randomized_MUSIC_trials(
        D_signals,
        t_samples,
        K_Rx,
        freq_range,
        min_freq_diff,
        AoA_range,
        min_angular_diff,
        snr,
        trials,
        sigma_2 = 1,
        cross_corr = 0,
        tail_percentile = 0.95,
):
    """ Run a set of randomized MUSIC trials and return the mean and tail of t

    Args:
        D_signals (int): The number of signals being received by the array
        t_samples (np.array): the time samples at which to sample the signals
        K_Rx (int): The number of receivers
        freq_range (np.array): [min frequency, max frequency]
        min_freq_diff (float): The minimum amount of separation when randomly
        AoA_range (np.array): [min AoA, max Aoa] (can be negative)
        min_angular_diff (float): The minimum amount of separation when random
        snr (float): the snr for the given simulation
        trials (int): the number of trials to run
        sigma_2 (int, optional): The noise variance. Defaults to 1.
        cross_corr (int, optional): The correlation of the noise between recei
```

```
        tail_percentile (float, optional): Tail percentail to report back. Def

    Returns:
        _type_: _description_
    """

    #compute the power and amplitude of the sample signals to achieve the give
    P_s_t,Amp_s_t = determine_signal_amplitude(snr,sigma_2)


    absolute_errors = np.zeros((D_signals,trials))

    #for each SNR trial
    for trial in range(0,trials):

        #compute a random set of frequencies and angles
        sample_freqs = generate_random_frequencies(freq_range,min_freq_diff,D_
        desired_angles = np.sort(generate_random_angles(AoA_range,min_angular_

        #compute s_t
        s_t = Amp_s_t * generate_signals(sample_freqs,t_samples,complex=True,v

        #generate A matrix (with each column as a mode vector) for the given a
        A = generate_mode_vector_matrix(thetas=desired_angles,K_rx=K_Rx,radian

        #generate noise samples
        n_t = generate_noise_samples(sigma_2,cross_corr,K_Rx,len(t_samples),ve

        #compute the samples received by each sensor
        x_t = np.matmul(A,s_t) + n_t

        #perform the MUSIC algorithm in transparent mode to understand what is
        P_mus, thetas, estimated_AoAs = MUSIC(
            x_t=x_t,
            K_rx=K_Rx,
            D_signals=D_signals,
            plot_spectrum=False)

        #determine the accuracy of the estimated AoAs
        if len(estimated_AoAs) != D_signals:
            #if MUSIC didn't detect all of the targets, give a result of NaN f
            absolute_errors[:,trial] = np.NaN
        else:
            absolute_errors[:,trial] = np.abs(estimated_AoAs - desired_angles)

    #flatten the absolute errors array
    absolute_errors = absolute_errors.flatten()

    #remove all Nan values
    absolute_errors = absolute_errors[np.logical_not(np.isnan(absolute_errors)

    #compute the mean of the absolute errors
    mean = np.mean(absolute_errors)

    #compute the tail of the absolute errors
    tail_idx = np.int_(np.floor(absolute_errors.size * tail_percentile))
    tail = np.sort(absolute_errors)[tail_idx]

    return mean,tail
```

```
def characterize_MUSIC(
        D_signals,
        t_samples,
        K_Rx,
        freq_range,
        min_freq_diff,
        AoA_range,
        min_angular_diff,
        SNRs,
        trials_per_SNR_step,
        sigma_2 = 1,
        cross_corrs = [0.0],
        tail_percentile = 0.95,
        error_bars = False,
):
    """Performs a series of randomized music trials for a given set of SNRs an
    in order to characterize the MUSIC algorithm

    Args:
        D_signals (int): The number of signals being received by the array
        t_samples (np.array): the time samples at which to sample the signals
        K_Rx (int): The number of receivers
        freq_range (np.array): [min frequency, max frequency]
        min_freq_diff (float): The minimum amount of separation when randomly
        AoA_range (np.array): [min AoA, max Aoa] (can be negative)
        min_angular_diff (float): The minimum amount of separation when random
        SNRs (np.array): the SNR values to evaluate for each cross correlation
        trials_per_SNR_step (int): number of trials to perform for each SNR tr
        sigma_2 (int, optional): the noise variance. Defaults to 1.
        cross_corrs (list, optional): the correlation between each receiver. D
        tail_percentile (float, optional): Tail percentile to use to generate
        error_bars (bool, optional): plots error bars on True. Defaults to Fal
    """
    #for a given cross_corr
    for cross_corr in cross_corrs:
        means = np.zeros(len(SNRs))
        tails = np.zeros(len(SNRs))
        #for each SNR level
        for i in range(len(SNRs)):
            snr = SNRs[i]
            mean,tail = run_randomized_MUSIC_trials(
                D_signals = D_signals,
                t_samples = t_samples,
                K_Rx = K_Rx,
                freq_range = freq_range,
                min_freq_diff = min_freq_diff,
                AoA_range = AoA_range,
                min_angular_diff = min_angular_diff,
                snr=snr,
                trials = trials_per_SNR_step,
                sigma_2 = sigma_2,
                cross_corr = cross_corr,
                tail_percentile = tail_percentile)

            means[i] = mean
            tails[i] = tail

        # plot average absolute error (in theta) vs SNR and include error
        plt.plot(SNRs, means, label = "Cross Correlation = {:0.2}".format(
        if error_bars:
```

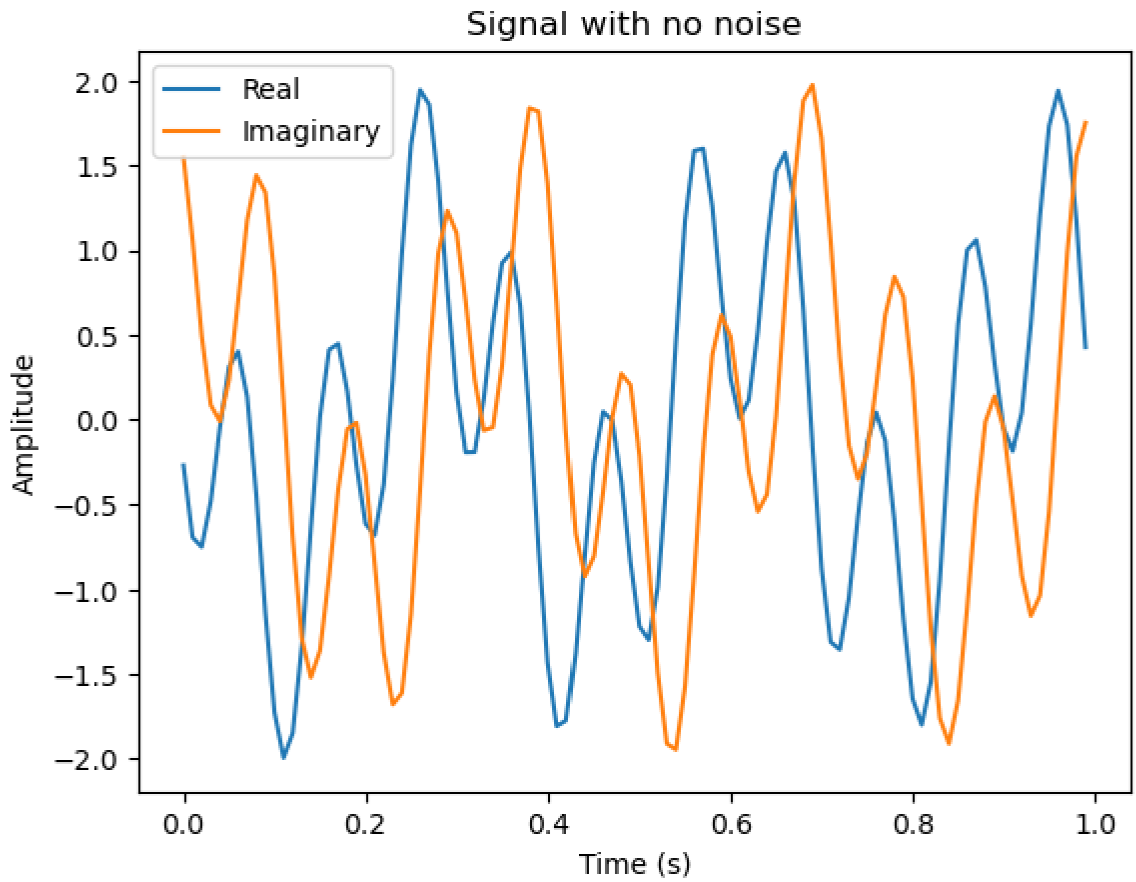
```
                    plt.errorbar(SNRs, means,
                            yerr = tails,
                            fmt ='o')
    plt.xlabel('SNR')
    plt.ylabel('Absolute Error (degrees)')
    plt.title("Absolute Error vs SNR for MUSIC")
    plt.legend()
    plt.show()
```

Initialize Global Experiment Parameters

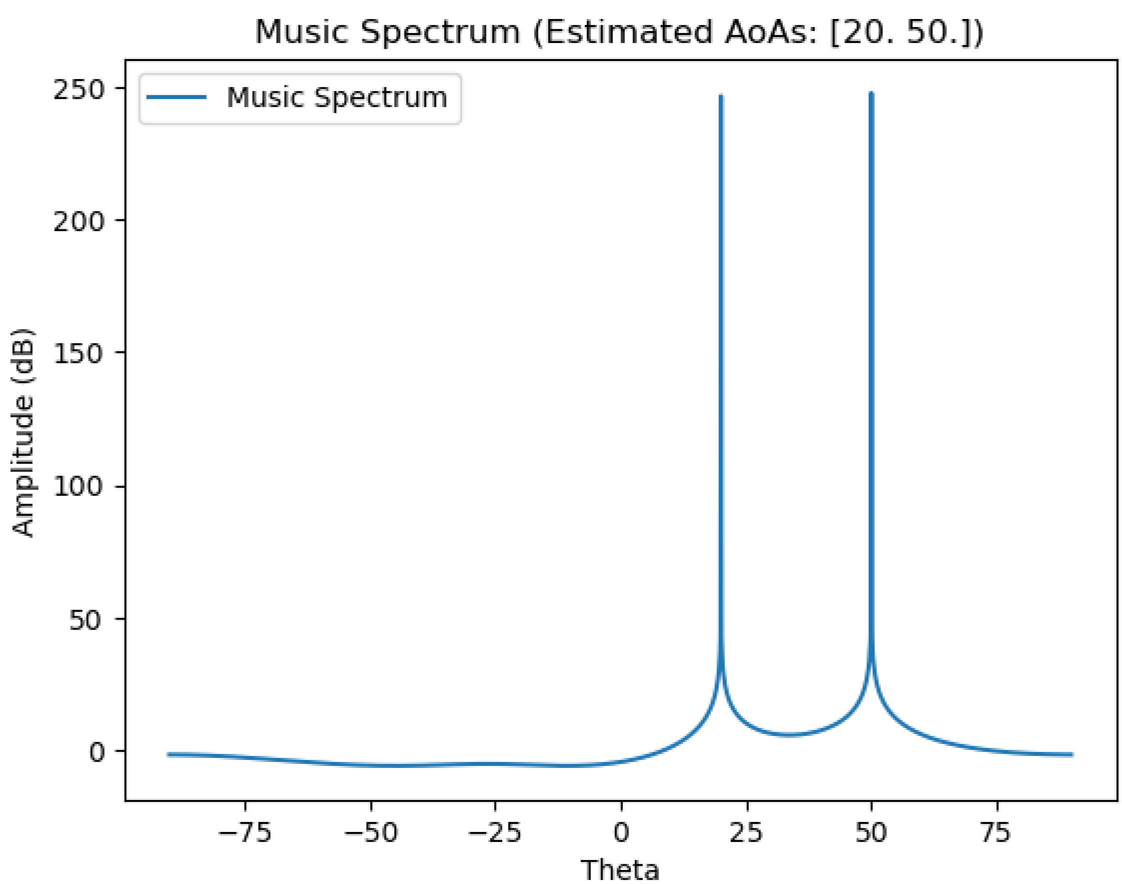
In [ ]:
```
# Sample rate
fs = 100

# Time points
t_samples = np.arange(0, 1, 1/fs)

#number of receivers
n_antennas = 4
```

Experiment #1: MUSIC with no noise

In [ ]:
```
#sample signal frequencies
sample_freqs = [10,3]

#sample signal AoAs
desired_angles = [50,20]

# generate sample signals using sinusoids of given frequencies
s_t = generate_signals(sample_freqs,t_samples,complex=True,verbose=False)

#generate A matrix (with each column as a mode vector) for the given angles
A = generate_mode_vector_matrix(thetas=desired_angles,K_rx=n_antennas,radians=

#compute the samples received by each sensor
x_t = np.matmul(A,s_t)

plot_signal(x_t[1,:],t_samples,title="Signal with no noise")

#perform the MUSIC algorithm in transparent mode to understand what is happeni
P_mus, thetas, estimated_AoAs = MUSIC(
    x_t=x_t,
    K_rx=n_antennas,
    D_signals=2,
    plot_spectrum=True,
    verbose=True,
    transparent_mode=True,
    s_t=s_t,
    A=A)
```



Signal with no noise
Real
Imaginary
Amplitude
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)



Music Spectrum (Estimated AoAs: [20. 50.])
Music Spectrum
Amplitude (dB)
250
200
150
100
50
0
−75
−50
−25
0
25
50
75
Theta

```
Sensor Covariance Matrix (magnitude):
[[1.01010101e+00 3.48745395e-16]
 [3.54747095e-16 1.01010101e+00]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[2.02020202 1.58841383 0.4776259  0.83733292]
 [1.58841383 2.02020202 1.58841383 0.4776259 ]
 [0.4776259  1.58841383 2.02020202 1.58841383]
 [0.83733292 0.4776259  1.58841383 2.02020202]]
         Eigen Values:
[4.79148496e+00 3.28932312e+00 2.07834556e-16 2.07834556e-16]
         Eigen Vectors:
[[-0.35133246-0.j          0.65906388+0.j         -0.47021288-0.j
  -0.47021288+0.j        ]
 [ 0.10366589-0.60482963j -0.04327955+0.25251077j  0.25381098+0.3160857j
  -0.39192966+0.48975571j]
 [ 0.57862406+0.2043519j   0.24157019+0.08531503j -0.09568819-0.54004743j
   0.00194505+0.50694029j]
 [-0.17128016+0.30675333j -0.32130411+0.57543797j -0.49454504+0.263035j
   0.17217265+0.31431621j]]



Rx:

         Covariance Matrix (magnitude):
[[2.02020202 1.58841383 0.4776259  0.83733292]
 [1.58841383 2.02020202 1.58841383 0.4776259 ]
 [0.4776259  1.58841383 2.02020202 1.58841383]
 [0.83733292 0.4776259  1.58841383 2.02020202]]
         Eigen Values:
[4.79148496e+00 3.28932312e+00 1.12887961e-16 5.56977171e-16]
         Eigen Vectors:
[[-0.35133246-0.j          0.65906388+0.j         -0.60636557+0.j
  -0.27298554+0.j        ]
 [ 0.10366589-0.60482963j -0.04327955+0.25251077j -0.15214159+0.70315643j
   0.10003547-0.17382913j]
 [ 0.57862406+0.2043519j   0.24157019+0.08531503j  0.20731465+0.12519696j
  -0.62196589-0.3351186j ]
 [-0.17128016+0.30675333j -0.32130411+0.57543797j -0.08724344+0.22018108j
  -0.36149252+0.50540315j]]



Estimated AoAs: [20. 50.]
```

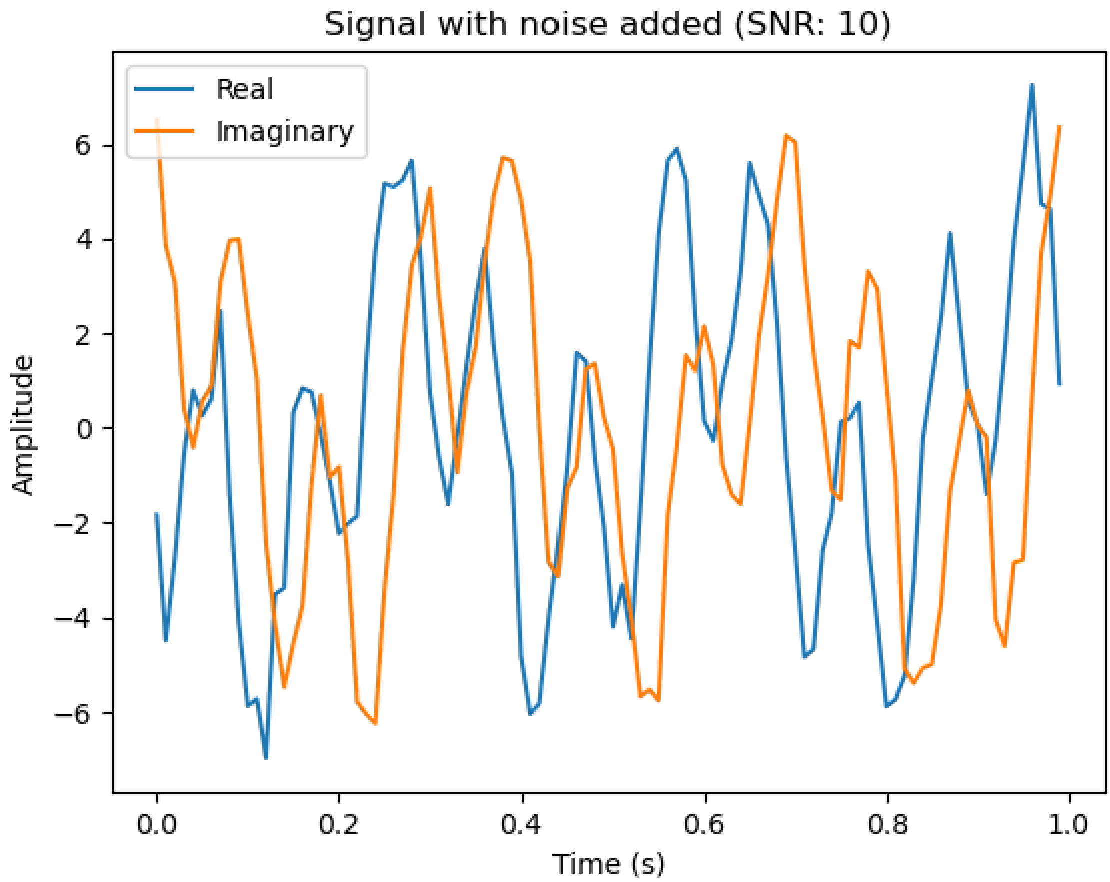
Experiment #2: MUSIC with white noise

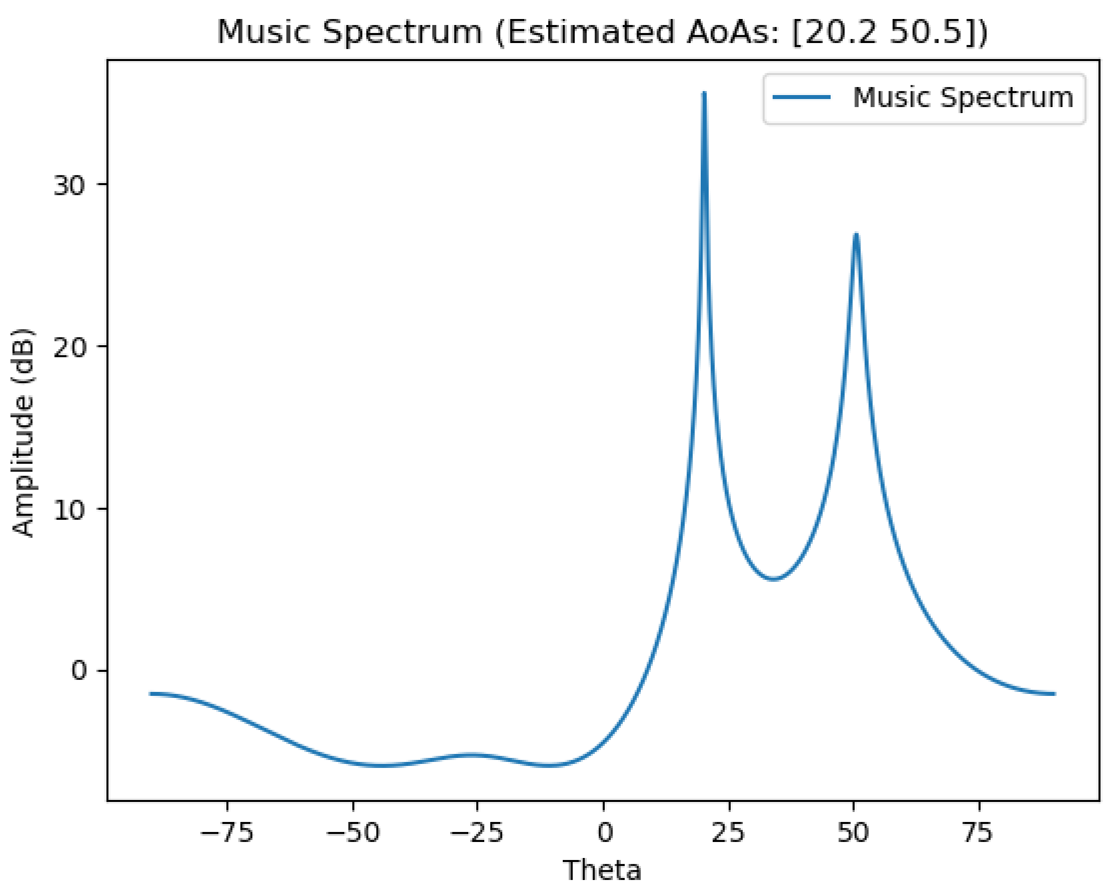
In [ ]:

```python
#sample signal frequencies
sample_freqs = [10,3]

#sample signal AoAs
desired_angles = [50,20]
```

```
#set noise variance
sigma_2 = 1
cross_corr = 0

#specify SNR
snr_dB = 10

#compute the power and amplitude of the sample signals to achieve the given SN
P_s_t,Amp_s_t = determine_signal_amplitude(snr_dB,sigma_2)


#compute s_t
s_t = Amp_s_t * generate_signals(sample_freqs,t_samples,complex=True,verbose=F

#generate A matrix (with each column as a mode vector) for the given angles
A = generate_mode_vector_matrix(thetas=desired_angles,K_rx=n_antennas,radians=


#generate noise samples
n_t = generate_noise_samples(sigma_2,cross_corr,n_antennas,len(t_samples),verb

#compute the samples received by each sensor
x_t = np.matmul(A,s_t) + n_t

plot_signal(x_t[1,:],t_samples,title="Signal with noise added (SNR: {})".forma


#perform the MUSIC algorithm in transparent mode to understand what is happeni
P_mus, thetas, estimated_AoAs = MUSIC(
    x_t=x_t,
    K_rx=n_antennas,
    D_signals=2,
    plot_spectrum=True,
    verbose=True,
    transparent_mode=True,
    s_t=s_t,
    A=A)
```

```
Set Noise Covariance Matrix:
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]



Actual Noise Coveriance Matrix:
[[0.89686985 0.07375062 0.05492203 0.01806894]
 [0.07375062 1.00294779 0.16513677 0.09998964]
 [0.05492203 0.16513677 1.05744675 0.12061423]
 [0.01806894 0.09998964 0.12061423 1.09778472]]
```


Signal with noise added (SNR: 10)
Real
Imaginary
Amplitude
Time (s)
6
4
2
0
−2
−4
−6
0.0
0.2
0.4
0.6
0.8
1.0


Music Spectrum (Estimated AoAs: [20.2 50.5])
Music Spectrum
Amplitude (dB)
Theta
30
20
10
0
−75
−50
−25
0
25
50
75

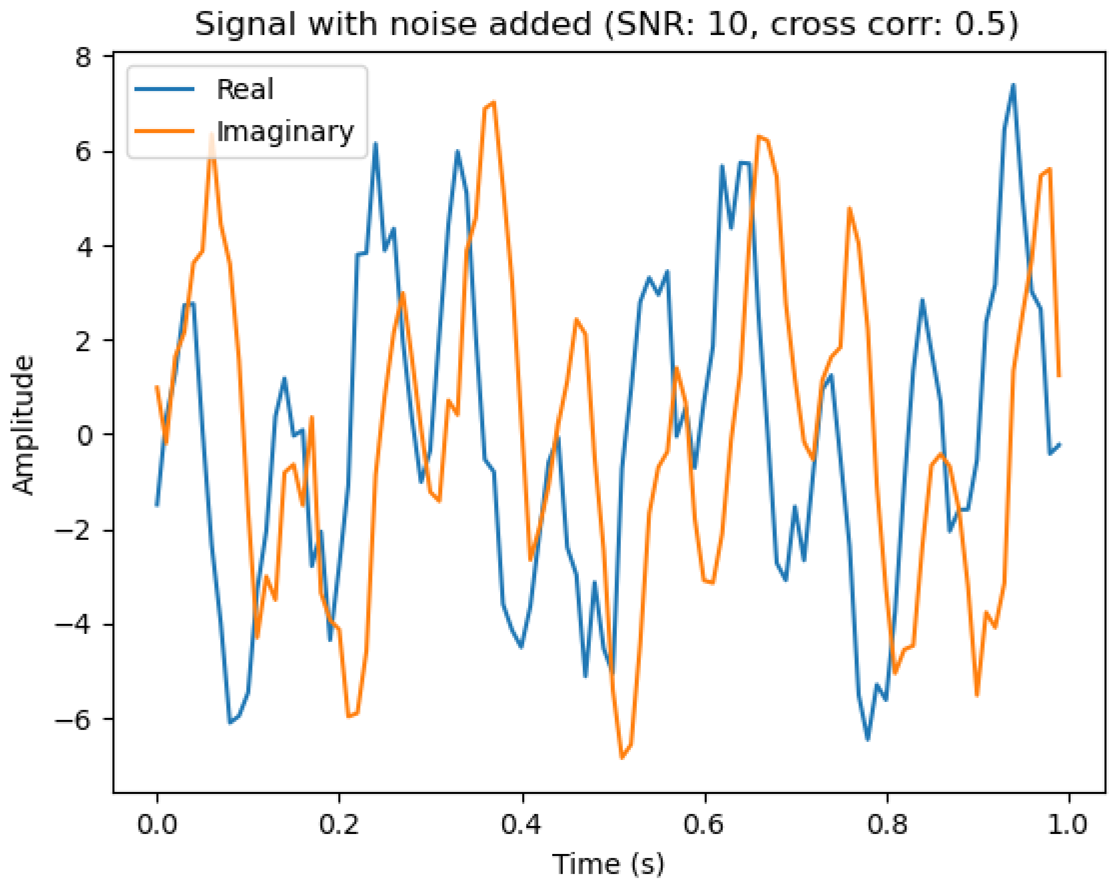
```
Sensor Covariance Matrix (magnitude):
[[1.01010101e+01 3.49388928e-15]
 [3.61836256e-15 1.01010101e+01]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[20.2020202  15.88413834  4.77625903  8.37332918]
 [15.88413834 20.2020202  15.88413834  4.77625903]
 [ 4.77625903 15.88413834 20.2020202  15.88413834]
 [ 8.37332918  4.77625903 15.88413834 20.2020202 ]]
         Eigen Values:
[4.79148496e+01 3.28932312e+01 1.44427945e-15 4.99699313e-15]
         Eigen Vectors:
[[-0.35133246-0.j          0.65906388+0.j         -0.58570382-0.j
   0.31488306+0.j        ]
 [ 0.10366589-0.60482963j -0.04327955+0.25251077j -0.11349591+0.27967918j
  -0.00485834-0.68313565j]
 [ 0.57862406+0.2043519j   0.24157019+0.08531503j -0.41008376-0.03706493j
  -0.62279753-0.0195046j ]
 [-0.17128016+0.30675333j -0.32130411+0.57543797j -0.24886588+0.57824955j
   0.01848925+0.21342842j]]



Rx:

         Covariance Matrix (magnitude):
[[21.66907908 17.05890207  4.79435719  8.18612022]
 [17.05890207 22.8205846  16.44606833  5.04562607]
 [ 4.79435719 16.44606833 21.72930935 16.32383124]
 [ 8.18612022  5.04562607 16.32383124 21.14800308]]
         Eigen Values:
[51.15753624 34.3165585   1.05793462  0.83494675]
         Eigen Vectors:
[[-0.37316165-0.j          0.64226969+0.j         -0.26125257+0.j
   0.61643096+0.j        ]
 [ 0.1793508 -0.6024535j  -0.0106789 +0.24916097j  0.29894764-0.00532236j
   0.24639647-0.62656094j]
 [ 0.50385072+0.31743457j  0.24313051+0.16937052j -0.28388669-0.6382362j
  -0.06862687-0.25480225j]
 [-0.25150799+0.21851074j -0.39674628+0.5292334j  -0.53647106+0.25809063j
   0.03375971-0.30975716j]]



Estimated AoAs: [20.2 50.5]
```

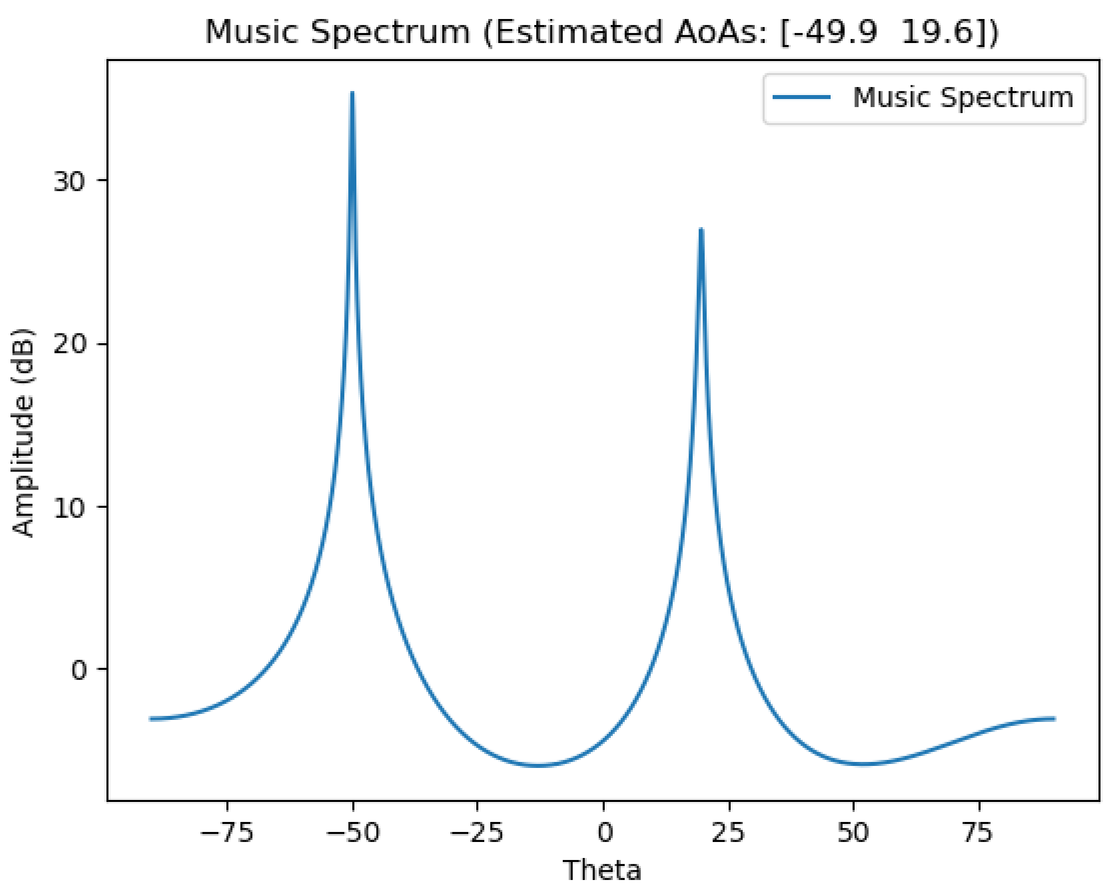
Experiment #3: MUSIC with Colored Noise:

In [ ]:
```python
#sample signal frequencies
sample_freqs = [10,3]

#sample signal AoAs
desired_angles = [-50,20]
```

```
#set noise variance
sigma_2 = 1
cross_corr = 0.5

#specify SNR
snr_dB = 10

#compute the power and amplitude of the sample signals to achieve the given SN
P_s_t,Amp_s_t = determine_signal_amplitude(snr_dB,sigma_2)


#compute s_t
s_t = Amp_s_t * generate_signals(sample_freqs,t_samples,complex=True,verbose=F

#generate A matrix (with each column as a mode vector) for the given angles
A = generate_mode_vector_matrix(thetas=desired_angles,K_rx=n_antennas,radians=


#generate noise samples
n_t = generate_noise_samples(sigma_2,cross_corr,n_antennas,len(t_samples),verb

#compute the samples received by each sensor
x_t = np.matmul(A,s_t) + n_t

plot_signal(x_t[1,:],t_samples,title="Signal with noise added (SNR: {}, cross



#perform the MUSIC algorithm in transparent mode to understand what is happeni
P_mus, thetas, estimated_AoAs = MUSIC(
    x_t=x_t,
    K_rx=n_antennas,
    D_signals=2,
    plot_spectrum=True,
    verbose=True,
    transparent_mode=True,
    s_t=s_t,
    A=A)
```

```
Set Noise Covariance Matrix:
[[1.  0.5 0.5 0.5]
 [0.5 1.  0.5 0.5]
 [0.5 0.5 1.  0.5]
 [0.5 0.5 0.5 1. ]]



Actual Noise Coveriance Matrix:
[[1.15455767 0.58252685 0.41501819 0.59572769]
 [0.58252685 1.13789094 0.48038233 0.59677118]
 [0.41501819 0.48038233 0.92390489 0.53779136]
 [0.59572769 0.59677118 0.53779136 1.11199454]]
```


Signal with noise added (SNR: 10, cross corr: 0.5)
Real
Imaginary
Amplitude
Time (s)
8
6
4
2
0
−2
−4
−6
0.0
0.2
0.4
0.6
0.8
1.0


Music Spectrum (Estimated AoAs: [-49.9 19.6])
Music Spectrum
Amplitude (dB)
Theta
30
20
10
0
−75
−50
−25
0
25
50
75

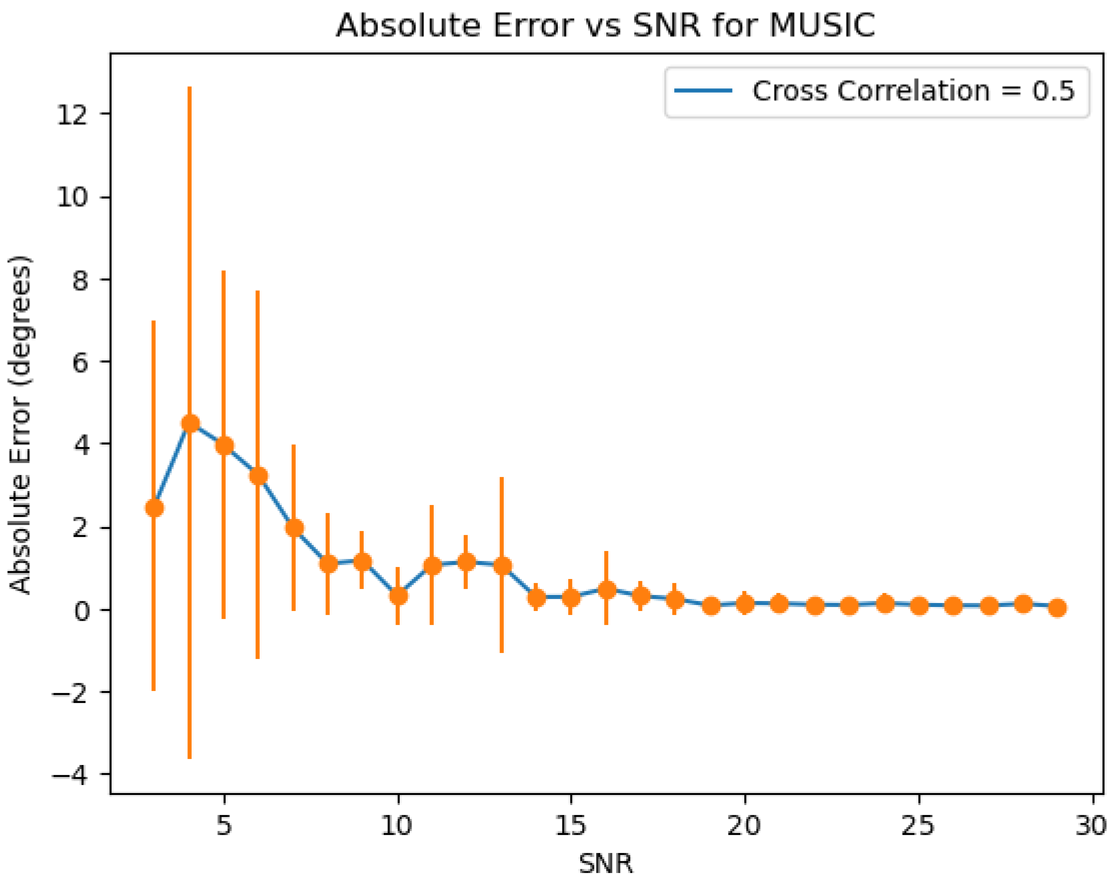
```
Sensor Covariance Matrix (magnitude):
[[1.01010101e+01 3.49388928e-15]
 [3.61836256e-15 1.01010101e+01]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[20.2020202   3.41279676 19.04894921  9.84880578]
 [ 3.41279676 20.2020202   3.41279676 19.04894921]
 [19.04894921  3.41279676 20.2020202   3.41279676]
 [ 9.84880578 19.04894921  3.41279676 20.2020202 ]]
         Eigen Values:
[4.68400494e+01 3.39680314e+01 3.26728609e-15 1.62726391e-14]
         Eigen Vectors:
[[-0.5663756 -0.j          0.39037987+0.j         -0.28899651+0.j
   0.66581022+0.j        ]
 [ 0.33285915-0.2615788j   0.46356535-0.36429483j  0.56707238-0.26455944j
   0.25748877-0.12375183j]
 [ 0.10008862-0.41134043j -0.13939114+0.57286444j  0.03488752+0.06008304j
   0.18201226-0.65971372j]
 [ 0.23475124+0.51543494j  0.16180457+0.35526853j -0.16345992-0.70240443j
   0.03387269-0.0747245j ]]



Rx:

         Covariance Matrix (magnitude):
[[20.99359506  3.02750501 18.83800326  9.32165352]
 [ 3.02750501 22.37053062  2.9768913  19.38918891]
 [18.83800326  2.9768913  21.25153257  3.05449568]
 [ 9.32165352 19.38918891  3.05449568 21.20064066]]
         Eigen Values:
[47.52180704 35.62552795  2.14937591  0.51958801]
         Eigen Vectors:
[[-0.53999563-0.j          0.42761067+0.j         -0.46290297+0.j
   0.55791996+0.j        ]
 [ 0.36759796-0.27991163j  0.4546021 -0.34227776j -0.47525493-0.22285981j
  -0.38695217-0.19348995j]
 [ 0.09246518-0.3909247j  -0.11921044+0.58938829j -0.25080462+0.4638376j
  -0.02722927-0.44525124j]
 [ 0.2336014 +0.52818973j  0.1614283 +0.32508952j -0.38088994+0.29506216j
  -0.21365038+0.50687118j]]



Estimated AoAs: [-49.9  19.6]
```

Experiment #4: Characterizing MUSIC Performance over different SNRs and correlations

Part 1: Comparing the effect of colored noise over SNR

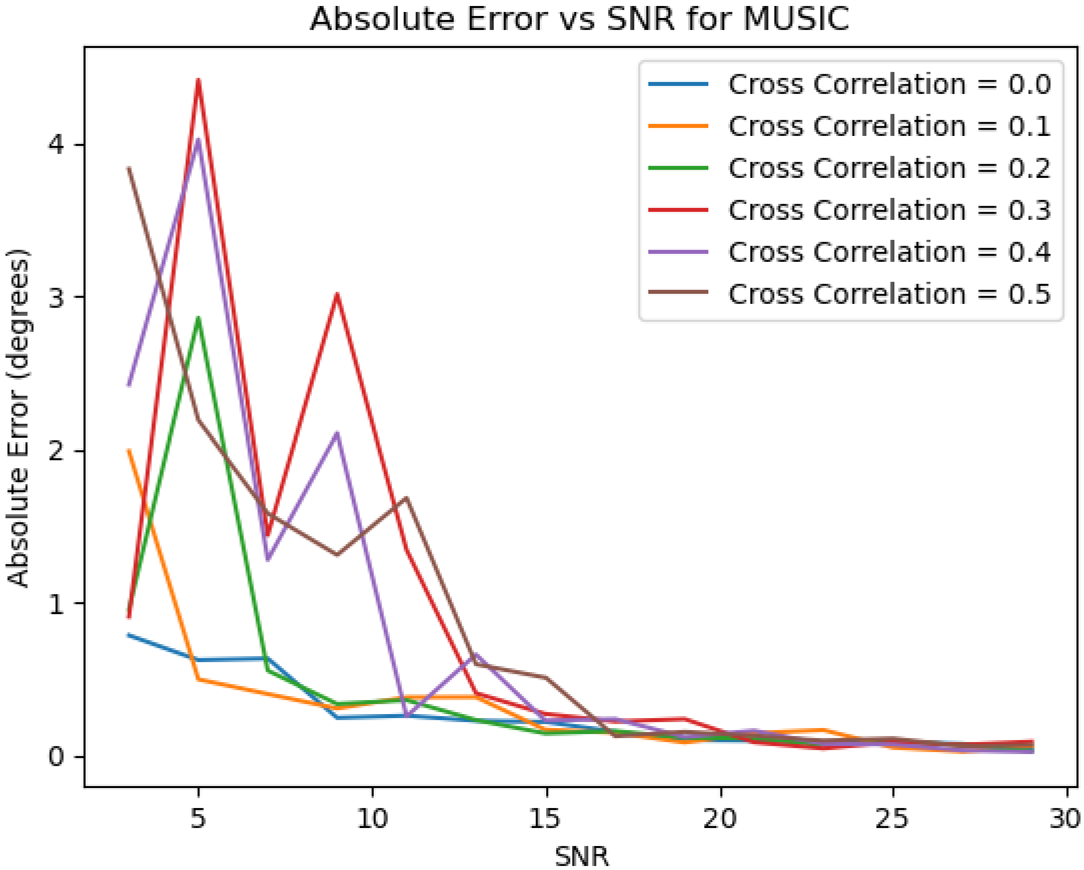
In [ ]:
```
characterize_MUSIC(
        D_signals = 2,
```

```
        t_samples = t_samples,
        K_Rx = n_antennas,
        freq_range = [1,10],
        min_freq_diff = 2,
        AoA_range = [-60,60],
        min_angular_diff = 5,
        SNRs = np.arange(3,30,1),
        trials_per_SNR_step = 20,
        sigma_2 = 1,
        cross_corrs = [0.5],
        tail_percentile = 0.85,
        error_bars = True)
```



Part 2: Sweeping over various cross correlation coefficients

In [ ]:
```
characterize_MUSIC(
        D_signals = 2,
        t_samples = t_samples,
        K_Rx = n_antennas,
        freq_range = [1,10],
        min_freq_diff = 2,
        AoA_range = [-60,60],
        min_angular_diff = 5,
        SNRs = np.arange(3,30,2),
        trials_per_SNR_step = 20,
        sigma_2 = 1,
        cross_corrs = np.arange(0,0.6,0.1),
        tail_percentile = 0.85,
        error_bars = False)
```


Absolute Error vs SNR for MUSIC
Cross Correlation = 0.0
Cross Correlation = 0.1
Cross Correlation = 0.2
Cross Correlation = 0.3
Cross Correlation = 0.4
Cross Correlation = 0.5
Absolute Error (degrees)
0
1
2
3
4
5
10
15
20
25
30
SNR