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

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)
    thetas = np.round_(thetas,2)
    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

    #find peaks in the MUSIC spectrum
    peak_indicies,properties = find_peaks(P_mus,height=-5)

    #identify the D_signals highest peaks in the spectrum
    highest_peaks_indicies = np.flip(np.argsort(properties["peak_heights"]))[:
    estimated_AoAs = np.sort(thetas[peak_indicies[highest_peaks_indicies]])

    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
```

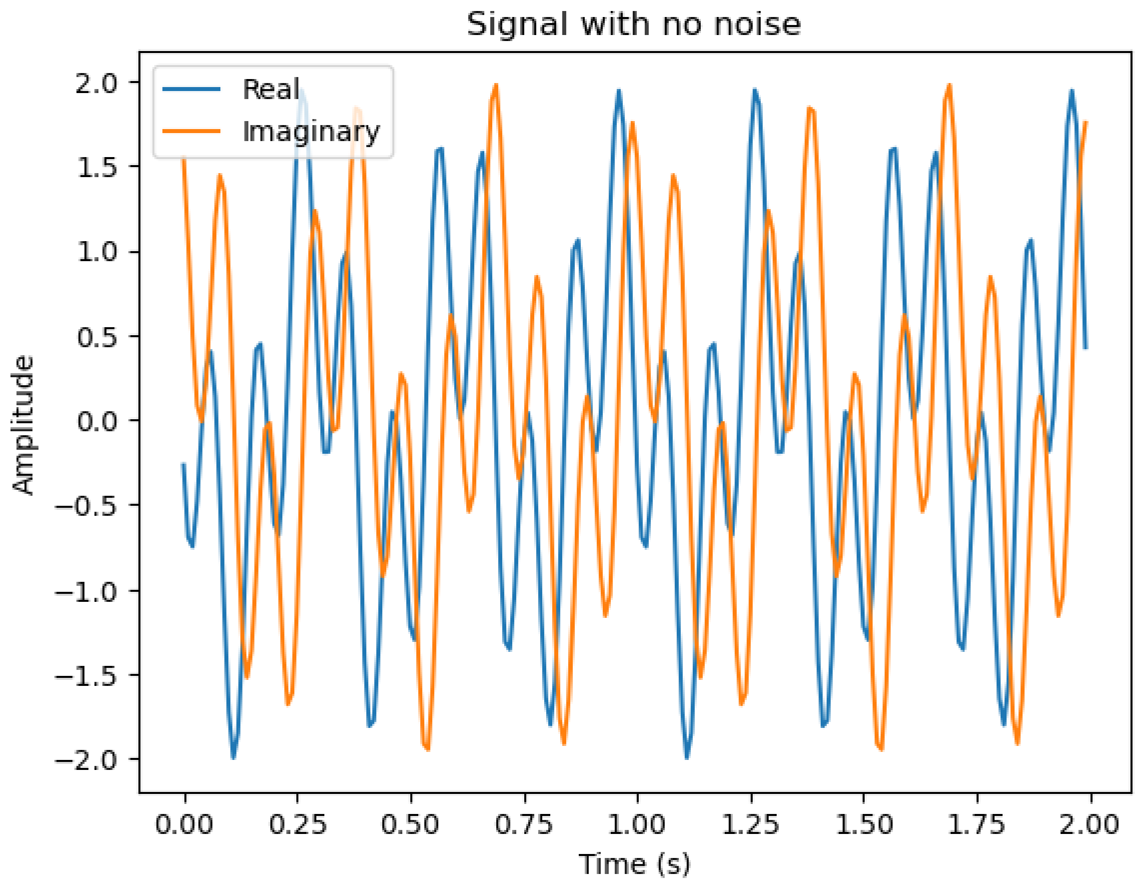
```
                tails[i] = tail

            # 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()
```

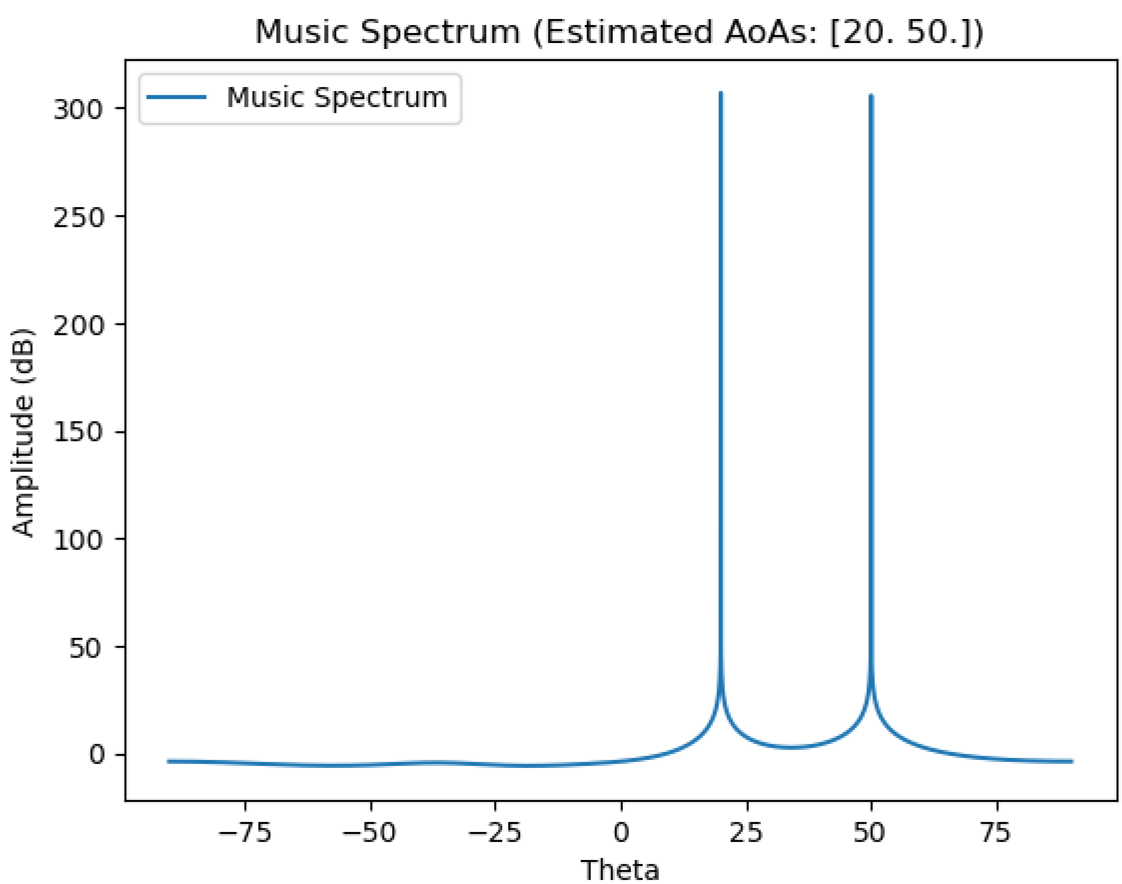
Initialize Global Experiment Parameters

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

# Time points
t_samples = np.arange(0, 2, 1/fs)

#number of receivers
n_antennas = 5
```

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
Time (s)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Music Spectrum (Estimated AoAs: [20. 50.])
Music Spectrum
Amplitude (dB)
Theta
300
250
200
150
100
50
0
−75
−50
−25
0
25
50
75

```
Sensor Covariance Matrix (magnitude):
[[1.00502513e+00 2.14016131e-16]
 [2.16006601e-16 1.00502513e+00]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[2.01005025 1.58043185 0.47522577 0.83312522 1.78533991]
 [1.58043185 2.01005025 1.58043185 0.47522577 0.83312522]
 [0.47522577 1.58043185 2.01005025 1.58043185 0.47522577]
 [0.83312522 0.47522577 1.58043185 2.01005025 1.58043185]
 [1.78533991 0.83312522 0.47522577 1.58043185 2.01005025]]
         Eigen Values:
[5.33021464e+00 4.72003661e+00 4.45201139e-16 2.42979636e-18
 8.91720145e-16]
         Eigen Vectors:
[[-5.96679156e-01-0.00000000e+00j  1.54285050e-01+0.00000000e+00j
   1.78004777e-02+0.00000000e+00j  4.45278735e-02-0.00000000e+00j
   7.86047403e-01+0.00000000e+00j]
 [ 6.40999420e-02-3.73985537e-01j -8.66792726e-02+5.05722678e-01j
   2.28664079e-01+3.47551936e-02j  5.85636781e-02-6.41674049e-01j
   5.71751387e-02-3.47588581e-01j]
 [ 8.87578942e-17+2.73361260e-16j -6.15328452e-01-2.17314742e-01j
  -1.53005603e-02-6.21573342e-01j -3.59488117e-01-1.83844903e-01j
   1.41487115e-01+6.71447302e-02j]
 [ 1.84982559e-01-3.31293581e-01j  2.50143030e-01-4.47992397e-01j
  -6.73393145e-01+3.05699491e-02j -5.29610639e-04-3.25172105e-01j
   1.06599436e-01-1.45821209e-01j]
 [ 4.64340204e-01+3.74718816e-01j  1.20065785e-01+9.68921245e-02j
   5.08594502e-02+3.20344969e-01j -5.07335914e-01-2.38084893e-01j
   3.56496353e-01+2.71659216e-01j]]



Rx:

         Covariance Matrix (magnitude):
[[2.01005025 1.58043185 0.47522577 0.83312522 1.78533991]
 [1.58043185 2.01005025 1.58043185 0.47522577 0.83312522]
 [0.47522577 1.58043185 2.01005025 1.58043185 0.47522577]
 [0.83312522 0.47522577 1.58043185 2.01005025 1.58043185]
 [1.78533991 0.83312522 0.47522577 1.58043185 2.01005025]]
         Eigen Values:
[5.33021464e+00 4.72003661e+00 9.83941179e-16 5.26322549e-16
 6.22085308e-16]
         Eigen Vectors:
[[-5.96679156e-01-0.00000000e+00j  1.54285050e-01+0.00000000e+00j
   7.45834997e-01+0.00000000e+00j  1.94033299e-01+0.00000000e+00j
  -1.62022663e-01-0.00000000e+00j]
 [ 6.40999420e-02-3.73985537e-01j -8.66792726e-02+5.05722678e-01j
   1.57053565e-01-5.06404894e-01j -2.24306995e-01-1.69404384e-02j
   1.35737823e-01-4.92564910e-01j]
 [ 1.48624539e-15+6.84503662e-16j -6.15328452e-01-2.17314742e-01j
   1.69275578e-02+2.61309299e-02j -1.47136611e-01-1.38971192e-01j
  -6.84226610e-01-2.53076279e-01j]
 [ 1.84982559e-01-3.31293581e-01j  2.50143030e-01-4.47992397e-01j
   2.41123427e-01-1.61338655e-01j -6.28025228e-01+1.83560160e-01j
  -8.51820601e-02+2.70593092e-01j]
```

```
[ 4.64340204e-01+3.74718816e-01j  1.20065785e-01+9.68921245e-02j
  2.72713277e-01+5.57301570e-02j  1.19636801e-01+6.54496229e-01j
 -1.97041175e-01-2.47362533e-01j]]
```

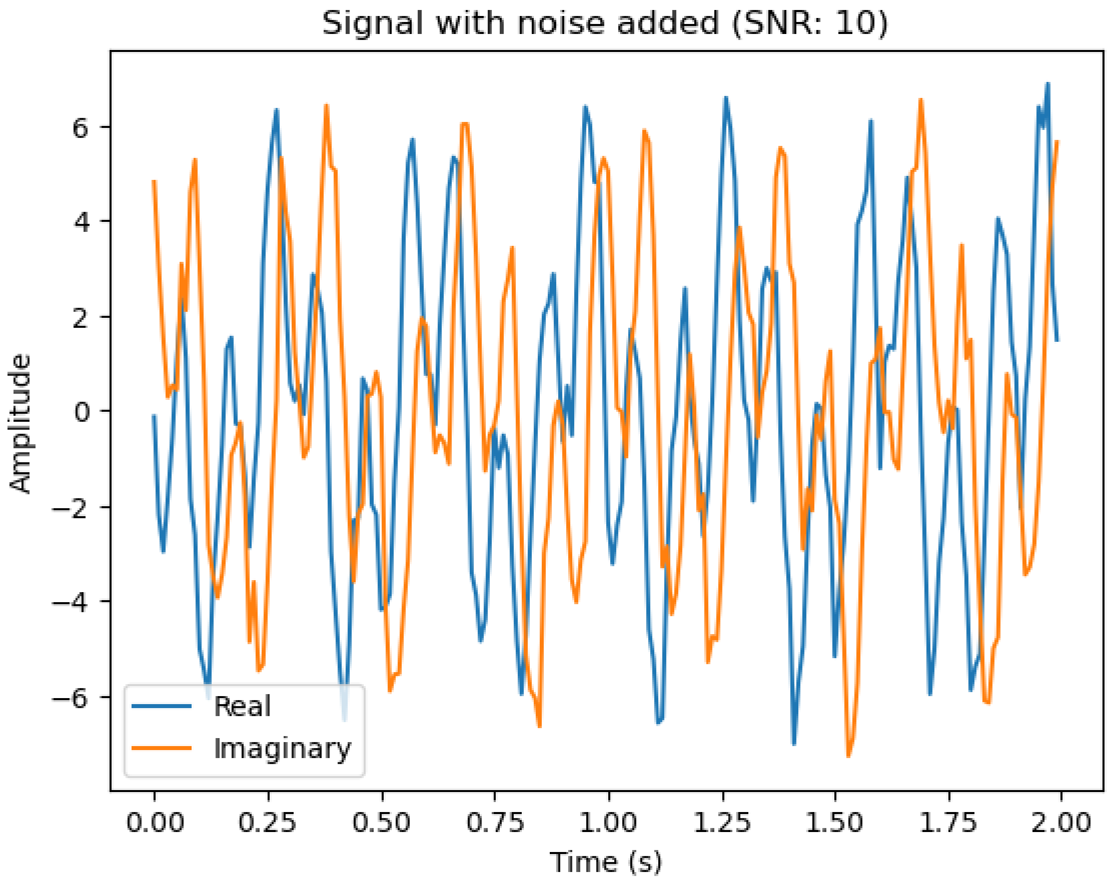
```
Estimated AoAs: [20. 50.]
```

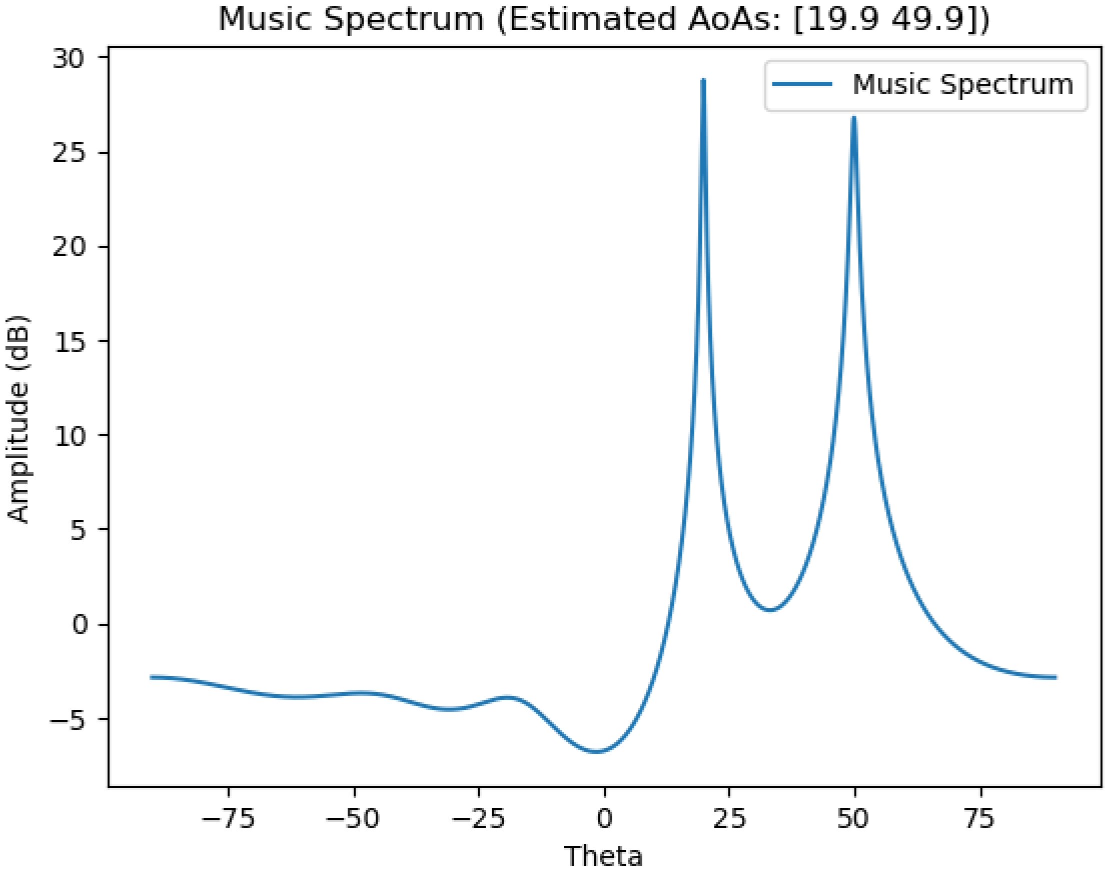
Experiment #2: MUSIC with white noise

In [ ]:

```
#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.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]



Actual Noise Coveriance Matrix:
[[0.98845674 0.03378153 0.0548053  0.08243931 0.03228502]
 [0.03378153 0.90126645 0.06716751 0.07226898 0.03202026]
 [0.0548053  0.06716751 1.03562192 0.04703914 0.07490607]
 [0.08243931 0.07226898 0.04703914 0.97471927 0.06734457]
 [0.03228502 0.03202026 0.07490607 0.06734457 1.01738597]]
```




Music Spectrum (Estimated AoAs: [19.9 49.9])
Music Spectrum
Amplitude (dB)
Theta
30
25
20
15
10
5
0
−5
−75
−50
−25
0
25
50
75

```
Sensor Covariance Matrix (magnitude):
[[1.00502513e+01 2.20463508e-15]
 [2.08628605e-15 1.00502513e+01]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[20.10050251 15.80431855  4.75225773  8.33125215 17.85339914]
 [15.80431855 20.10050251 15.80431855  4.75225773  8.33125215]
 [ 4.75225773 15.80431855 20.10050251 15.80431855  4.75225773]
 [ 8.33125215  4.75225773 15.80431855 20.10050251 15.80431855]
 [17.85339914  8.33125215  4.75225773 15.80431855 20.10050251]]
         Eigen Values:
[5.33021464e+01 4.72003661e+01 1.48516742e-14 3.82760209e-16
 6.35265025e-15]
         Eigen Vectors:
[[-5.96679156e-01-0.00000000e+00j  1.54285050e-01+0.00000000e+00j
  -7.74253669e-01+0.00000000e+00j  7.85340699e-02-0.00000000e+00j
   1.20556059e-01+0.00000000e+00j]
 [ 6.40999420e-02-3.73985537e-01j -8.66792726e-02+5.05722678e-01j
  -1.44984867e-01+4.29787586e-01j  7.89894427e-02-1.31689677e-02j
  -5.54414380e-01+2.70612737e-01j]
 [ 7.35904717e-16+4.63666466e-16j -6.15328452e-01-2.17314742e-01j
  -1.10375922e-01-1.01006272e-02j -1.38540738e-01-4.84328547e-01j
   1.68860749e-01+5.28752024e-01j]
 [ 1.84982559e-01-3.31293581e-01j  2.50143030e-01-4.47992397e-01j
  -1.76460915e-01+1.91499933e-01j -6.99788808e-01+1.57504477e-01j
  -8.20061650e-02+6.09062132e-02j]
 [ 4.64340204e-01+3.74718816e-01j  1.20065785e-01+9.68921245e-02j
  -3.07776527e-01-1.41319986e-01j  1.01835927e-01+4.56912877e-01j
   1.01554996e-01+5.25375131e-01j]]



Rx:

         Covariance Matrix (magnitude):
[[20.97219549 15.98973936  4.4270326   8.63521399 17.50043294]
 [15.98973936 21.75715694 15.84564608  4.81528078  8.21888489]
 [ 4.4270326  15.84564608 20.78445181 16.05298491  5.05295455]
 [ 8.63521399  4.81528078 16.05298491 21.83912548 15.99573141]
 [17.50043294  8.21888489  5.05295455 15.99573141 20.44021671]]
         Eigen Values:
[54.24668455 48.66732501  1.0294734   0.97232791  0.87733557]
         Eigen Vectors:
[[-5.85289883e-01-0.j          1.91983853e-01+0.j
   2.77262895e-01+0.j          6.43205842e-01+0.j
   3.60540546e-01+0.j        ]
 [ 5.38066117e-02-0.34849298j -9.73274324e-02+0.53661612j
  -1.52708202e-04+0.03820229j  3.59561596e-01-0.42412608j
  -5.02168259e-01-0.12421003j]
 [-4.95578587e-02-0.00284892j -6.02838939e-01-0.22261734j
   4.03253691e-01-0.29758878j  4.68172486e-02-0.11238487j
  -1.53077887e-01+0.54326325j]
 [ 1.91337402e-01-0.37473998j  2.33781822e-01-0.42965703j
  -4.83756684e-01-0.46912629j  3.09376645e-01-0.09785871j
   6.21406902e-03+0.15579469j]
 [ 4.72264266e-01+0.36132472j  1.01183067e-01+0.0569415j
```

```
 -1.09764019e-01+0.45204544j  3.64081639e-01-0.15636777j
  1.47667261e-01+0.48757093j]]
```

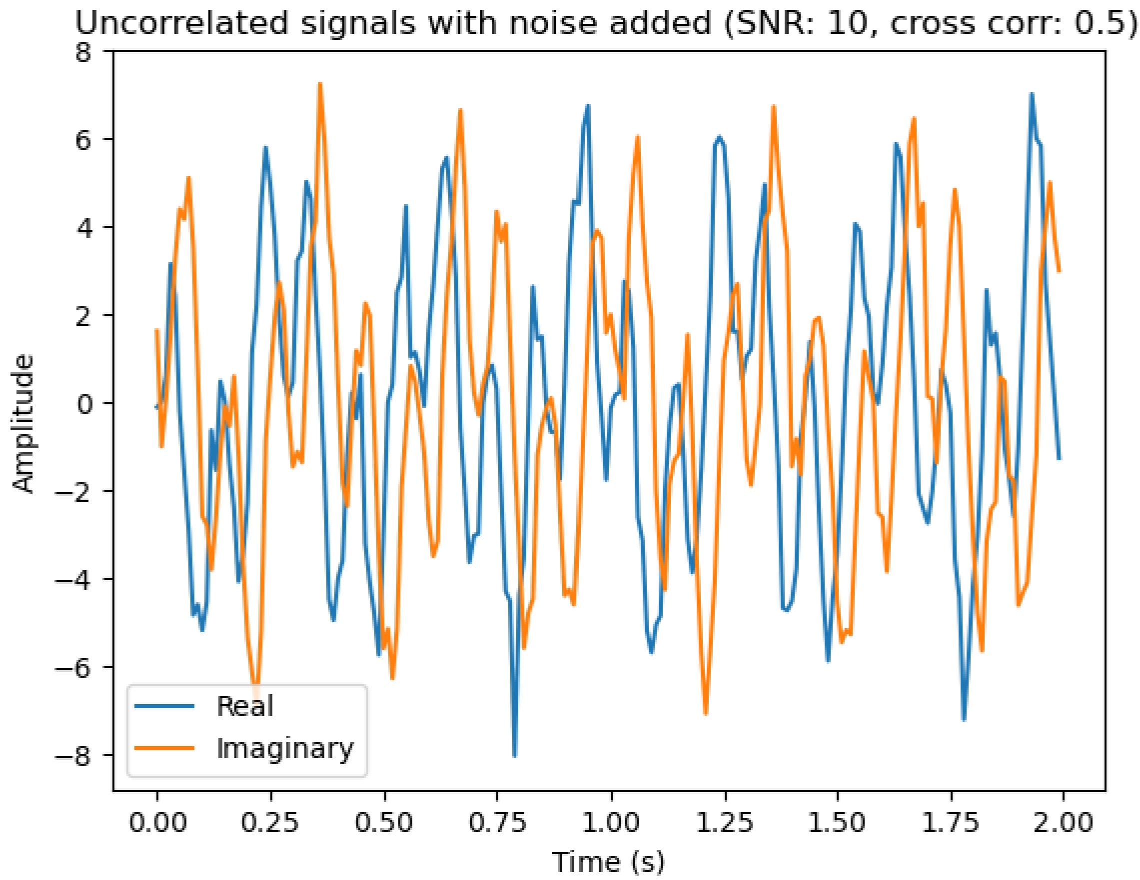
```
Estimated AoAs: [19.9 49.9]
```

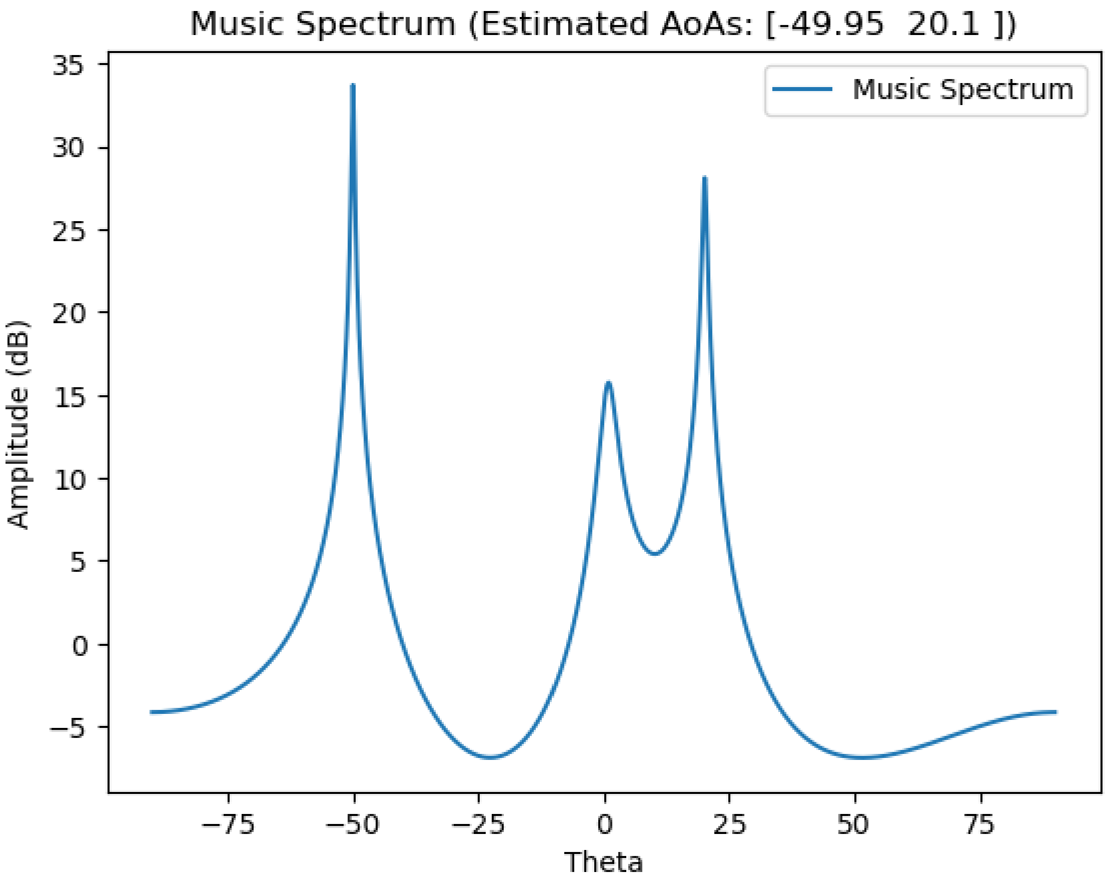
Experiment #3: MUSIC with Colored Noise:

Part 1: Colored Noise with uncorrelated signals

In [ ]:
```
#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="Uncorrelated signals with noise added (S



#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]
 [0.5 1.  0.5 0.5 0.5]
 [0.5 0.5 1.  0.5 0.5]
 [0.5 0.5 0.5 1.  0.5]
 [0.5 0.5 0.5 0.5 1. ]]



Actual Noise Coveriance Matrix:
[[1.0201962  0.37510816 0.43918809 0.45189814 0.45543265]
 [0.37510816 0.87358335 0.36783477 0.42773443 0.48262204]
 [0.43918809 0.36783477 0.92801661 0.45742296 0.49705665]
 [0.45189814 0.42773443 0.45742296 0.9639754  0.53502123]
 [0.45543265 0.48262204 0.49705665 0.53502123 1.05896744]]
```




Music Spectrum (Estimated AoAs: [-49.95 20.1])
Music Spectrum
Amplitude (dB)
35
30
25
20
15
10
5
0
−5
−75
−50
−25
0
25
50
75
Theta

```
Sensor Covariance Matrix (magnitude):
[[1.00502513e+01 2.20463508e-15]
 [2.08628605e-15 1.00502513e+01]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[20.10050251  3.39564703 18.95322585  9.7993143  15.64236211]
 [ 3.39564703 20.10050251  3.39564703 18.95322585  9.7993143 ]
 [18.95322585  3.39564703 20.10050251  3.39564703 18.95322585]
 [ 9.7993143  18.95322585  3.39564703 20.10050251  3.39564703]
 [15.64236211  9.7993143  18.95322585  3.39564703 20.10050251]]
         Eigen Values:
[5.69906438e+01 4.35118688e+01 1.60775772e-14 8.79970486e-15
 5.33735685e-15]
         Eigen Vectors:
[[-5.59986940e-01-0.00000000e+00j  2.26338277e-01+0.00000000e+00j
   7.30677801e-01-0.00000000e+00j -2.63762210e-01+0.00000000e+00j
   1.78115296e-01+0.00000000e+00j]
 [-7.88834980e-02+6.19909374e-02j -5.26721741e-01+4.13926554e-01j
   3.64645178e-01+3.63874318e-03j  5.45076783e-01+1.86041802e-01j
  -2.67376913e-01-7.05231949e-02j]
 [ 1.40408935e-01-5.77047328e-01j  4.16649991e-16-9.51280558e-16j
   1.04183538e-01-3.38958321e-01j -6.16149397e-03-1.85215199e-01j
   4.92603433e-03-6.97988438e-01j]
 [ 4.15834640e-02+9.13033301e-02j -2.77661553e-01-6.09651578e-01j
   2.29382155e-01-1.20588585e-03j  1.64330181e-01-6.31062306e-01j
  -2.14067649e-01+1.32200246e-01j]
 [ 4.97384100e-01+2.57282782e-01j  2.01035153e-01+1.03989821e-01j
   3.20136206e-01+2.28645650e-01j -3.69824775e-01+4.84995272e-02j
  -5.52650845e-01-1.89404494e-01j]]



Rx:

         Covariance Matrix (magnitude):
[[20.9608681   2.92828816 19.40138008  9.84207137 15.40600039]
 [ 2.92828816 20.67525362  2.75231715 19.02812013  9.49058208]
 [19.40138008  2.75231715 22.01114554  3.90942433 19.32428184]
 [ 9.84207137 19.02812013  3.90942433 21.74091629  2.74093008]
 [15.40600039  9.49058208 19.32428184  2.74093008 21.51325778]]
         Eigen Values:
[58.91323924 44.25709608  2.65306684  0.56437354  0.51366562]
         Eigen Vectors:
[[-0.5600493 -0.j          0.18610018+0.j          0.53140671+0.j
  -0.46024477+0.j         -0.39685409+0.j        ]
 [-0.04368981+0.05530452j -0.50430516+0.44009751j  0.35136096+0.05324194j
   0.01631921-0.27463835j  0.2767312 +0.51813239j]
 [ 0.12963219-0.58711163j -0.01123376-0.02247758j  0.39590181+0.00373956j
   0.15223817+0.65288067j  0.16536816+0.06584457j]
 [ 0.0488301 +0.12702421j -0.30431338-0.60108577j  0.35753627-0.08001484j
  -0.1603851 -0.17871717j  0.45314776-0.36101119j]
 [ 0.49448849+0.23839971j  0.23202525+0.09475928j  0.54416278-0.06494428j
   0.4054866 -0.20241047j -0.3306236 -0.1442201j ]]
```

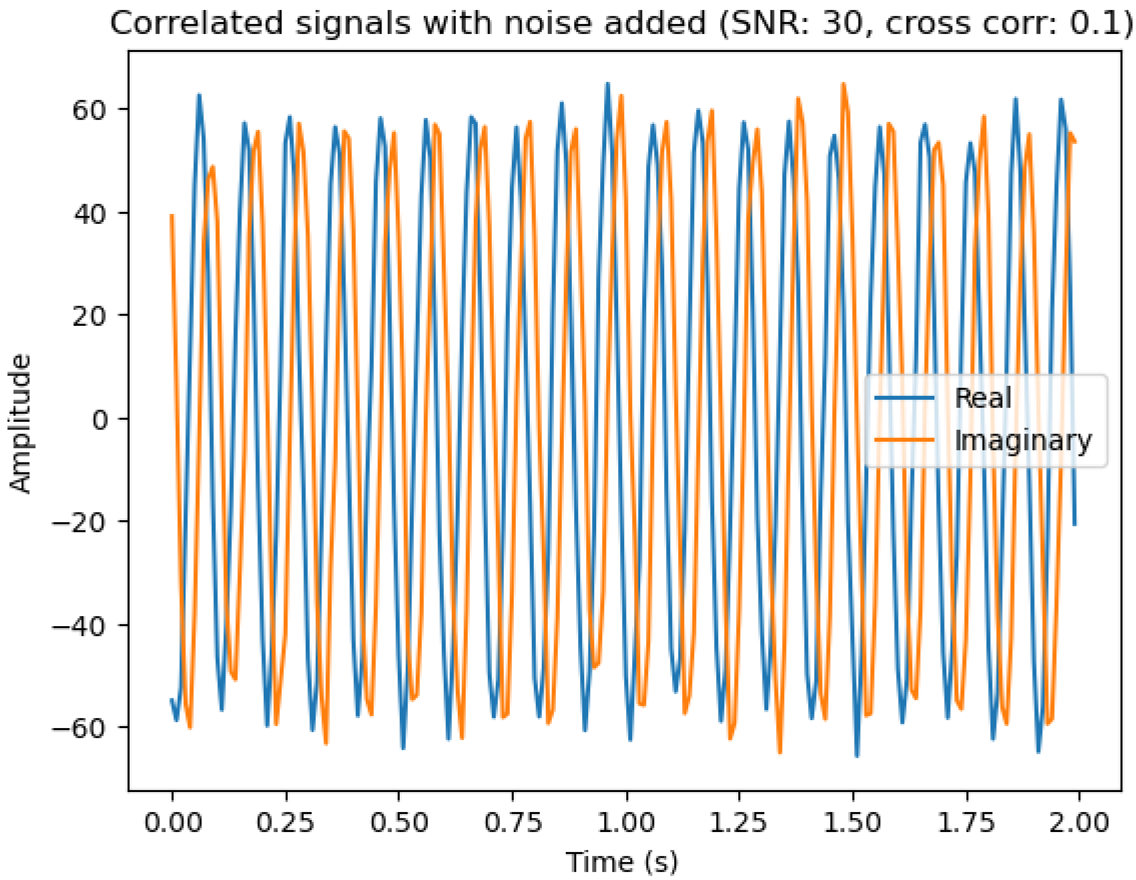
```
Estimated AoAs: [-49.95  20.1 ]
```

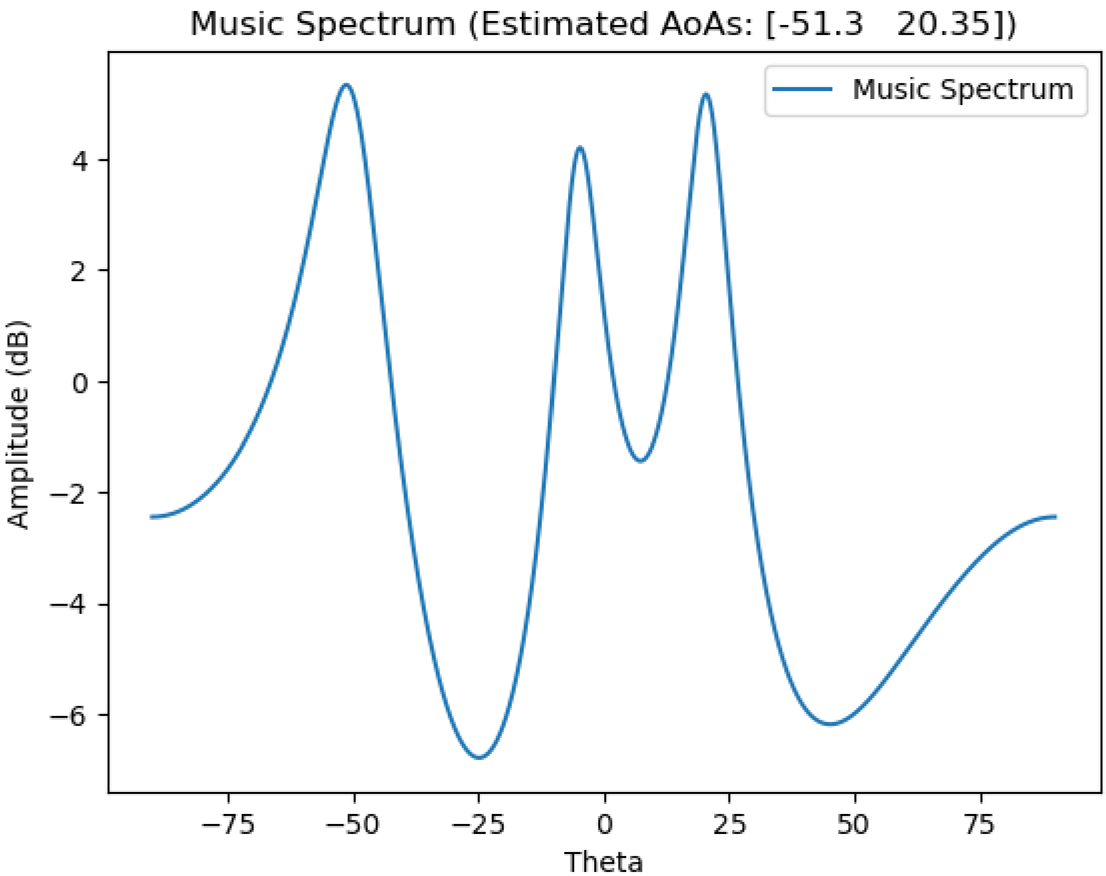
Part 2: Colored Noise with Correlated Signals

In [ ]:

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

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

#set noise variance
sigma_2 = 30
cross_corr = 0.1

#specify SNR
snr_dB = 30

#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="Correlated signals with noise added (SNR



#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:
[[30.   0.1  0.1  0.1  0.1]
 [ 0.1 30.   0.1  0.1  0.1]
 [ 0.1  0.1 30.   0.1  0.1]
 [ 0.1  0.1  0.1 30.   0.1]
 [ 0.1  0.1  0.1  0.1 30. ]]


Actual Noise Coveriance Matrix:
[[31.32697719  1.33164805  1.55213781  1.3538118   1.05193923]
 [ 1.33164805 30.68783447  1.94258372  3.41096502  1.95105446]
 [ 1.55213781  1.94258372 34.7827505   2.69776561  4.66325956]
 [ 1.3538118   3.41096502  2.69776561 28.89137009  1.63281276]
 [ 1.05193923  1.95105446  4.66325956  1.63281276 29.41923759]]
```




Music Spectrum (Estimated AoAs: [-51.3   20.35])
Music Spectrum
Amplitude (dB)
4
2
0
−2
−4
−6
Theta
−75
−50
−25
0
25
50
75

```
Sensor Covariance Matrix (magnitude):
[[30150.75376884 30150.75376884]
 [30150.75376884 30150.75376884]]



A Rs A^H:

         Covariance Matrix (magnitude):
[[120603.01507538  20373.88216366 113719.35509127  58795.88577458
   93854.17265398]
 [ 20373.88216366   3441.82999206  19211.00180546   9932.59121864
   15855.10820791]
 [113719.35509127  19211.00180546 107228.59386468  55439.99217703
   88497.25672417]
 [ 58795.88577458   9932.59121864  55439.99217703  28663.92835915
   45755.40015631]
 [ 93854.17265398  15855.10820791  88497.25672417  45755.40015631
   73038.02246617]]
         Eigen Values:
[3.32975390e+05 3.71737283e-11 1.37648053e-11 5.30417680e-12
 8.94898584e-12]
         Eigen Vectors:
[[ 6.01828866e-01+0.j          6.64204002e-01-0.j
   3.95369159e-01+0.j         -1.61509664e-01+0.j
   1.19301787e-01+0.j        ]
 [-7.99387769e-02+0.06282023j  3.49405453e-01-0.19598883j
  -3.28799684e-01+0.17199472j  5.90428197e-01+0.17321402j
   3.46935941e-01+0.43875175j]
 [-1.34165949e-01+0.55139014j  1.96772550e-01-0.57946442j
  -1.52473048e-01+0.21907463j -3.08164404e-01+0.13637324j
  -3.30594901e-01-0.09681055j]
 [-1.21608841e-01-0.2670122j   1.47986230e-04+0.10687476j
   6.91048353e-02+0.58245181j -1.12948662e-01+0.59192337j
   2.30719069e-01-0.37696947j]
 [-4.15989563e-01-0.21517968j  2.06497932e-02-0.10949952j
   4.69561094e-01+0.26502241j -3.04457993e-01-0.16098996j
   1.52216419e-02+0.59888663j]]



Rx:

         Covariance Matrix (magnitude):
[[120715.128052    20533.67545956 113846.03754515  58731.71117938
   93846.47238125]
 [ 20533.67545956   3523.62983283  19369.77708609   9995.28944897
   15965.38979561]
 [113846.03754515  19369.77708609 107432.29179206  55401.86131264
   88530.18635638]
 [ 58731.71117938   9995.28944897  55401.86131264  28610.32523682
   45671.70793206]
 [ 93846.47238125  15965.38979561  88530.18635638  45671.70793206
   73005.17294822]]
         Eigen Values:
[3.33164398e+05 3.72959812e+01 3.25071389e+01 2.77852976e+01
 2.45617349e+01]
         Eigen Vectors:
[[ 0.60188746+0.j          0.44684992-0.j         -0.35764801+0.j
  -0.53787701+0.j          0.14433594+0.j        ]
```

```
 [-0.08012961+0.06376664j -0.10341981-0.24825012j -0.3190139 -0.66943738j
   0.12312449+0.1863155j   0.32267284-0.46182478j]
 [-0.13511245+0.55148364j  0.55030144-0.44682072j -0.04216083+0.12456006j
   0.2874133 +0.1890244j  -0.17366007+0.09665815j]
 [-0.12097722-0.26675779j  0.24192162-0.33122887j  0.23420348-0.2069224j
  -0.21242771-0.52781651j -0.45578084-0.34182747j]
 [-0.41515011-0.21612529j  0.11001446-0.21255383j -0.41418467+0.1888433j
   0.0267448 -0.46695556j  0.46396472+0.28709086j]]
```

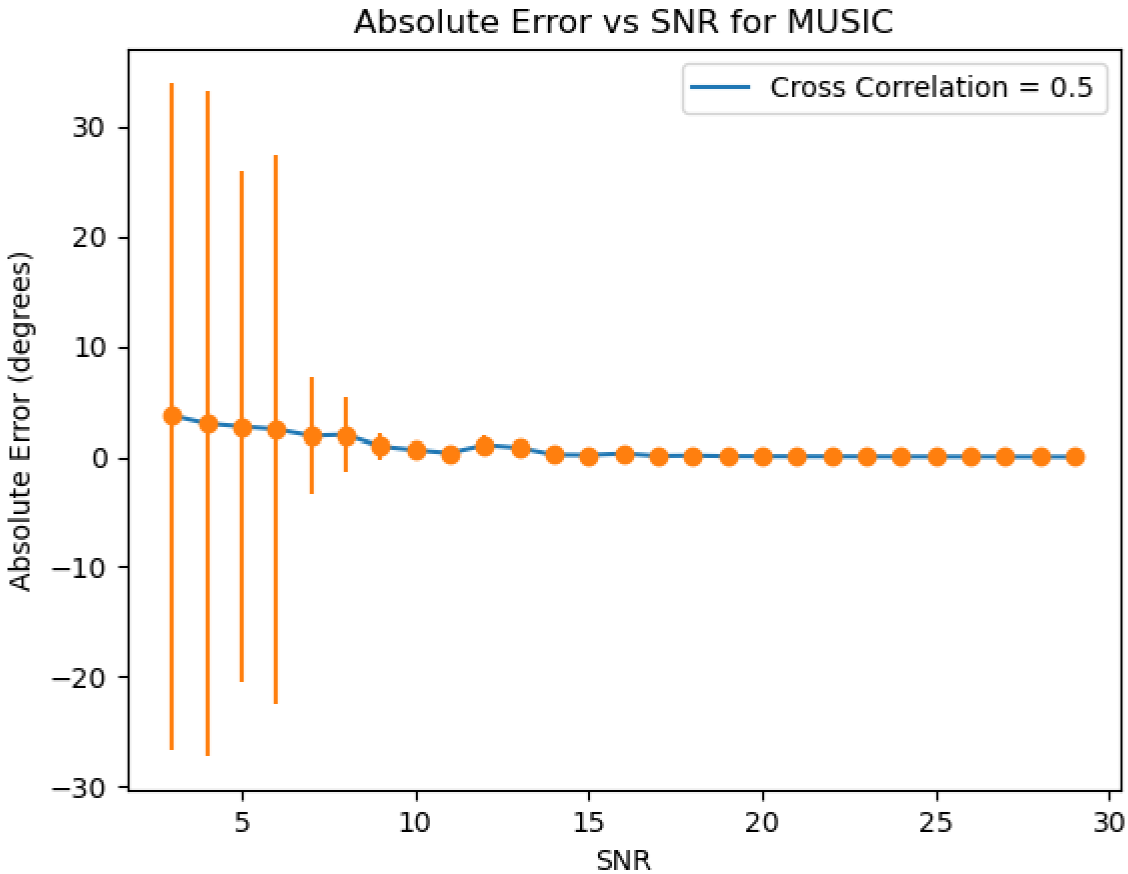
```
Estimated AoAs: [-51.3   20.35]
```

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

Part 1: Comparing the effect of colored noise over SNR

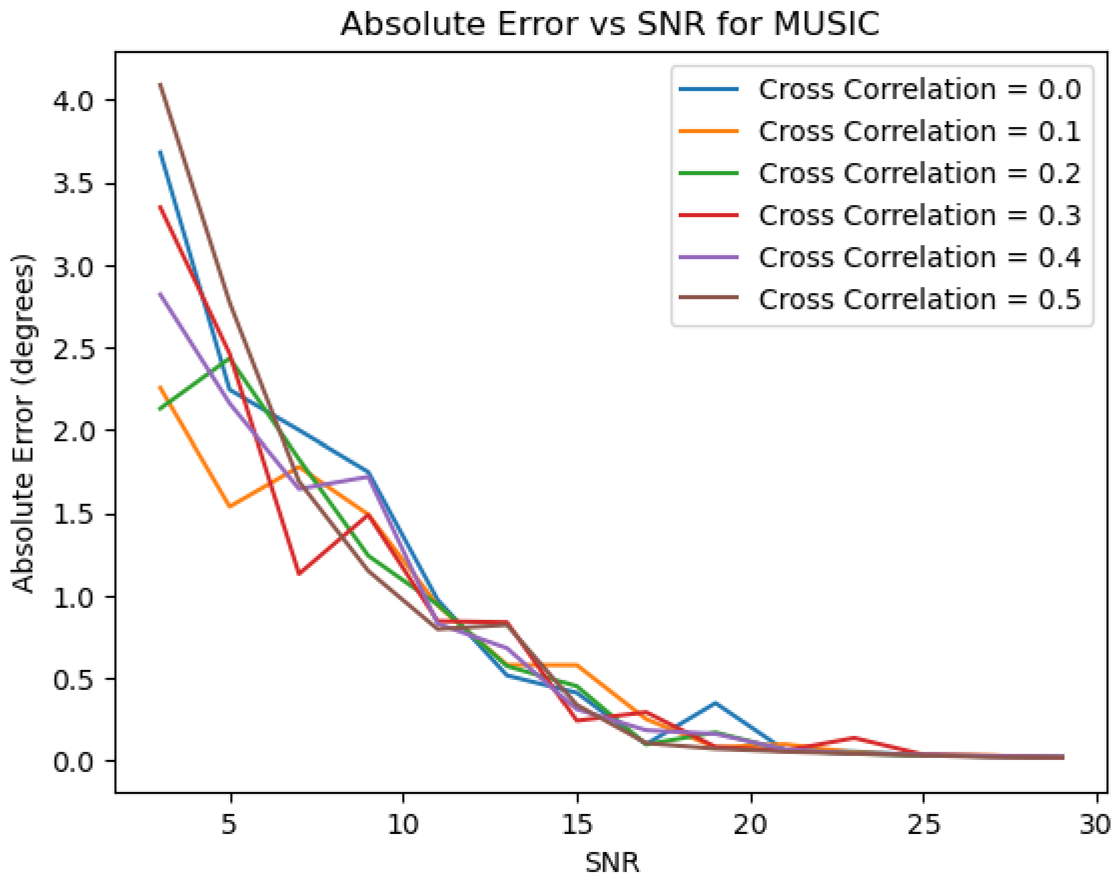
In [ ]:

```
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 = 200,
        sigma_2 = 1,
        cross_corrs = [0.5],
        tail_percentile = 0.95,
        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 = 300,
        sigma_2 = 1,
        cross_corrs = np.arange(0,0.6,0.1),
        tail_percentile = 0.95,
        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)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
5
10
15
20
25
30
SNR