

E1 244: Detection and Estimation

February-May 2021

Solution – Homework 3

Analysis and Algorithms for Spectrum Sensing in Cognitive Radio

Part A: Derivation and Modelling

Consider the OFDM transmission of the sequence s of length N_d , where the N_d -point IFFT of the sequence along with an N_c length cyclic prefix is transmitted over an AWGN channel. The transmitted sequence is defined using:

$$x[n] = \frac{1}{\sqrt{N_d}} \sum_{k=0}^{N_d-1} s[k] e^{j2\pi nk/N_d}, \quad n = 0, 1, \dots, N_d - 1, \quad (1)$$

The transmitted vector has entries defined by $x[n]$ with the last N_c points are prefixed to itself to form an $N_d + N_c$ length transmission vector $\mathbf{x}_i = [x[0] \ x[1] \ \dots \ x[N_d + N_c - 1]]^T$. This forms one OFDM symbol block. $K + 1$ such OFDM symbol blocks $\mathbf{x} = [\mathbf{x}_0^T \ \mathbf{x}_1^T \ \dots \ \mathbf{x}_K^T]^T$ are transmitted over an AWGN channel to give measurements $\mathbf{y} = \mathbf{x} + \mathbf{w}$, where $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma_w^2 \mathbf{I})$.

Energy Detector

Consider the data symbols $\mathbf{s} = [s[0] \ s[1] \ \dots \ s[N_d - 1]]^T$ where the entries are QPSK with variance $\sigma_s^2 = 1$, i.e., $s[k] \in \{\pm 1/\sqrt{2} \pm j1/\sqrt{2}\}$. Suppose the number of data points N_d is large, using the central limit theorem, the OFDM symbol blocks \mathbf{x}_i can be assumed to be zero-mean Gaussians with identity covariance of size $(K + 1)(N_d + N_c)$ as $\mathbb{E}[\mathbf{x}_i \mathbf{x}_i^H] = \sigma_s^2 \mathbf{I}$.

Given measurements $\mathbf{y} \in \mathbb{C}^{(K+1)(N_d+N_c)}$, the signal detection problem is to select between one of the two following hypothesis:

$$\begin{aligned} \mathcal{H}_0 : \mathbf{y} &= \mathbf{w}, \\ \mathcal{H}_1 : \mathbf{y} &= \mathbf{x} + \mathbf{w}. \end{aligned} \quad (2)$$

The Neymann-Pearson detector uses the likelihood ratio test (LRT). Let the measurements have the density function $p_Y(\mathbf{y}; \mathcal{H}_0)$ and $p_Y(\mathbf{y}; \mathcal{H}_1)$ under the hypothesis \mathcal{H}_0 and \mathcal{H}_1 respectively. The LRT compares the ratio of the likelihoods to a threshold, and decides on the hypothesis \mathcal{H}_1 if:

$$\begin{aligned} L(\mathbf{y}) &= \frac{p_Y(\mathbf{y}; \mathcal{H}_1)}{p_Y(\mathbf{y}; \mathcal{H}_0)} > \gamma, \\ &= \frac{\frac{1}{(2\pi(\sigma_s^2 + \sigma_w^2))^{N/2}} \exp\left(-\frac{1}{2(\sigma_s^2 + \sigma_w^2)} \sum_{n=0}^N |y[n]|^2\right)}{\frac{1}{(2\pi\sigma_w^2)^{N/2}} \exp\left(-\frac{1}{2\sigma_w^2} \sum_{n=0}^N |y[n]|^2\right)}, \\ &= \left(\frac{\sigma_w^2}{\sigma_s^2 + \sigma_w^2}\right)^{N/2} \exp\left(\frac{1}{2} \frac{\sigma_s^2}{\sigma_w^2(\sigma_s^2 + \sigma_w^2)} \sum_{n=0}^{N-1} |y[n]|^2\right), \\ \implies T(\mathbf{y}) &= \sum_{n=0}^{N-1} |y[n]|^2 > \gamma'. \end{aligned} \quad (3)$$

where $N = N_d + N_c$ and γ' is a threshold that is set by P_{FA} . The test statistic $T(\mathbf{y})$ is the square of sum of Gaussian random variables and hence, has the chi-squared distribution. The hypotheses with this test statistic are:

$$\begin{aligned}\mathcal{H}_0 : \frac{T(\mathbf{y})}{\sigma_w^2} &\sim \chi^2 \\ \mathcal{H}_1 : \frac{T(\mathbf{y})}{\sigma_w^2 + \sigma_s^2} &\sim \chi^2.\end{aligned}\tag{4}$$

The probability of false alarm, $P_{FA} = p_Y(T(\mathbf{y}) > \gamma'; \mathcal{H}_0) = Q(\frac{\gamma'}{\sigma_w^2})$, where $Q(\cdot)$ is the CDF of the chi-squared distribution with N degrees of freedom. Hence, given a choice for P_{FA} , the threshold can be computed as $\gamma' = \sigma_w^2 Q^{-1}(P_{FA})$. Using this, the probability of detection:

$$\begin{aligned}P_D &= p_Y(T(\mathbf{y}) > \gamma'; \mathcal{H}_1), \\ &= Q\left(\frac{\gamma'}{\sigma_w^2 + \sigma_s^2}\right).\end{aligned}\tag{5}$$

Cyclostationarity Detector

Under the same transmission settings, consider test function for LRT defined as:

$$\begin{aligned}T(\mathbf{y}) &= \sum_{n=0}^{N_c-1} \hat{R}[n], \\ &= \sum_{n=0}^{N_c-1} \frac{1}{K} \sum_{k=0}^{K-1} \hat{r}[n + k(N_c + N_d), N_d], \\ &= \frac{1}{K} \sum_{n=0}^{N_c-1} \sum_{k=0}^{K-1} y[n + k(N_c + N_d)] y^*[n + k(N_c + N_d) + N_d].\end{aligned}\tag{6}$$

Suppose the number of data samples are high, using the central limit theorem, the test statistic $T(\mathbf{y})$ can be taken to be a complex Gaussian random variable.

a) Distribution of the test statistic: Let $T(\mathbf{y}) = \bar{T}(\mathbf{y}) + j\tilde{T}(\mathbf{y})$ be defined as in (6) be approximated as a complex Gaussian random variable.

Under \mathcal{H}_0 , $\mathbf{y} = \mathbf{w}$, i.e., $y[m] = w[m]$, $\forall m$. Therefore, $\mathbb{E}[y[n + kN]y^*[n + kN + N_d]] = \mathbb{E}[w[n + kN]w^*[n + kN + N_d]] = 0$.

$$\begin{aligned}\mathbb{E}[T(\mathbf{y})] &= \frac{1}{K} \sum_{n=0}^{N_c-1} \sum_{k=0}^{K-1} \mathbb{E}[y[n + kN]y^*[n + kN + N_d]], \\ &= 0, \\ \implies \mathbb{E}[\bar{T}(\mathbf{y})] &= \mathbb{E}[\tilde{T}(\mathbf{y})] = 0.\end{aligned}\tag{7}$$

To find the variance, consider the equations for $T^2(\mathbf{y}) = \bar{T}^2(\mathbf{y}) + j2\bar{T}(\mathbf{y})\tilde{T}(\mathbf{y}) - \tilde{T}^2(\mathbf{y})$ and $|T(\mathbf{y})|^2 = \bar{T}^2(\mathbf{y}) + \tilde{T}^2(\mathbf{y})$.

$$\begin{aligned}T^2(\mathbf{y}) &= \frac{1}{K^2} \sum_{n_1=0}^{N_c-1} \sum_{k_1=0}^{K-1} \sum_{n_2=0}^{N_c-1} \sum_{k_2=0}^{K-1} y[n_1 + k_1N] y^*[n_1 + k_1N + N_d] \\ &\quad y[n_2 + k_2N] y^*[n_2 + k_2N + N_d].\end{aligned}\tag{8}$$

We have, $\mathbb{E}[y[n_1 + k_1 N]y^*[n_1 + k_1 N + N_d]y[n_2 + k_2 N]y^*[n_2 + k_2 N + N_d]] = \mathbb{E}[w[n_1 + k_1 N]w^*[n_1 + k_1 N + N_d]w[n_2 + k_2 N]w^*[n_2 + k_2 N + N_d]] = \sigma_w^4 \delta(n_1 - n_2, k_1 - k_2)$, and hence:

$$\mathbb{E}[T^2(\mathbf{y})] = \frac{1}{K^2} N_c K \sigma_w^4 = \frac{1}{K} N_c \sigma_w^4. \quad (9)$$

Since the quantity is real, $\mathbb{E}[\bar{T}(\mathbf{y})\tilde{T}(\mathbf{y})] = \text{cov}(\bar{T}(\mathbf{y}), \tilde{T}(\mathbf{y})) = 0$. Since the measurements are noise-only and real, $\mathbb{E}[|T(\mathbf{y})|^2] = \mathbb{E}[T^2(\mathbf{y})] = \frac{1}{K} N_c \sigma_w^4$. Using this and (9):

$$\text{var}(\bar{T}(\mathbf{y})) = \text{var}(\tilde{T}(\mathbf{y})) = \frac{1}{2K} N_c \sigma_w^4. \quad (10)$$

Under \mathcal{H}_1 , $\mathbf{y} = \mathbf{x} + \mathbf{w}$, i.e., $y[m] = x[m] + w[m]$, $\forall m$. Therefore, $\mathbb{E}[y[n + kN]y^*[n + kN + N_d]] = \mathbb{E}[(x[n + kN] + w[n + kN])(x[n + kN + N_d] + w[n + kN + N_d])^*] = \sigma_s^2 = 1$.

$$\begin{aligned} \mathbb{E}[T(\mathbf{y})] &= \frac{1}{K} \sum_{n=0}^{N_c-1} \sum_{k=0}^{K-1} \mathbb{E}[y[n + kN]y^*[n + kN + N_d]], \\ &= \frac{1}{K} N_c K = N_c, \\ \implies \mathbb{E}[\bar{T}(\mathbf{y})] &= N_c, \\ \mathbb{E}[\tilde{T}(\mathbf{y})] &= 0, \end{aligned} \quad (11)$$

since $\mathbb{E}[T(\mathbf{y})]$ is real. To find the variance, similar to the case in \mathcal{H}_0 , using (13), we have:

$$\begin{aligned} &\mathbb{E}[y[n_1 + k_1 N]y^*[n_1 + k_1 N + N_d]y[n_2 + k_2 N]y^*[n_2 + k_2 N + N_d]] = \\ &\mathbb{E}[(x[n_1 + k_1 N] + w[n_1 + k_1 N])(x[n_1 + k_1 N + N_d] + w[n_1 + k_1 N + N_d])^* \\ &(x[n_2 + k_2 N] + w[n_2 + k_2 N])(x[n_2 + k_2 N + N_d] + w[n_2 + k_2 N + N_d])^*] = \begin{cases} \sigma_s^4, & n_1 \neq n_2, k_1 \neq k_2, \\ \sigma_s^4 + \sigma_w^4, & n_1 = n_2, k_1 = k_2, \end{cases} \end{aligned}$$

and hence:

$$\mathbb{E}[T^2(\mathbf{y})] = \frac{1}{K^2} (2N_c K + N_c^2 K^2 - N_c K) = N_c^2 + \frac{N_c}{K}. \quad (12)$$

Since the quantity is real, $\mathbb{E}[\bar{T}(\mathbf{y})\tilde{T}(\mathbf{y})] = \text{cov}(\bar{T}(\mathbf{y}), \tilde{T}(\mathbf{y})) = 0$. In this case, the measurements are not all real. Hence:

$$\begin{aligned} |T(\mathbf{y})|^2 &= \frac{1}{K^2} \sum_{n_1=0}^{N_c-1} \sum_{k_1=0}^{K-1} \sum_{n_2=0}^{N_c-1} \sum_{k_2=0}^{K-1} y[n_1 + k_1 N]y^*[n_1 + k_1 N + N_d] \\ &\quad y^*[n_2 + k_2 N]y[n_2 + k_2 N + N_d], \end{aligned} \quad (13)$$

with the mean of each term in the sum:

$$\begin{aligned} &\mathbb{E}[y[n_1 + k_1 N]y^*[n_1 + k_1 N + N_d]y^*[n_2 + k_2 N]y[n_2 + k_2 N + N_d]] = \\ &\mathbb{E}[(x[n_1 + k_1 N] + w[n_1 + k_1 N])(x[n_1 + k_1 N + N_d] + w[n_1 + k_1 N + N_d])^* \\ &(x[n_2 + k_2 N] + w[n_2 + k_2 N])^*(x[n_2 + k_2 N + N_d] + w[n_2 + k_2 N + N_d])] \\ &= \begin{cases} \sigma_s^4, & n_1 \neq n_2, k_1 \neq k_2, \\ 2\sigma_s^4 + 2\sigma_s^2 \sigma_w^2 + \sigma_w^4, & n_1 = n_2, k_1 = k_2, \end{cases} \end{aligned}$$

and hence,

$$\mathbb{E}[|T(\mathbf{y})|^2] = N_c^2 + \frac{N_c}{K} (1 + 2\sigma_w^2 + \sigma_w^4). \quad (14)$$

Using (12) and (14):

$$\begin{aligned}\text{var}(\bar{T}(\mathbf{y})) &= N_c^2 + \frac{N_c}{K} \left(1 + \sigma_w^2 + \frac{\sigma_w^4}{2} \right), \\ \text{var}(\tilde{T}(\mathbf{y})) &= \frac{N_c}{K} \left(\sigma_w^2 + \frac{\sigma_w^4}{2} \right).\end{aligned}\tag{15}$$

b) Neymann-Pearson detector: Using the test statistic defined in (6), the detector chooses hypotheses \mathcal{H}_1 if $|T(\mathbf{y})| > \gamma$ or equivalently, $|T(\mathbf{y})|^2 > \gamma^2$. The probability of false alarm:

$$\begin{aligned}P_{FA} &= p_Y(|T(\mathbf{y})|^2 > \gamma^2; \mathcal{H}_0), \\ &= p_Y(\bar{T}^2(\mathbf{y}) + \tilde{T}^2(\mathbf{y}) > \gamma^2; \mathcal{H}_0), \\ &= Q\left(\frac{\gamma^2}{\frac{N_c}{2K}\sigma_w^4}\right),\end{aligned}\tag{16}$$

where $Q(\cdot)$ is the CDF of the chi-squared distribution with 2 degrees of freedom. Therefore, given a choice for the probability of false alarm, the threshold is chosen as:

$$\gamma = \left(\frac{N_c}{2K}\sigma_w^4 Q^{-1}(P_{FA}) \right)^{1/2}.\tag{17}$$

Part B: Implementation

Energy Detector

a) Monte-Carlo simulations with exact parameters:

Figures 1(a) and 1(b) shows the variation of P_D and P_{FA} with the signal-to-noise ratio (SNR). The estimated results are averaged over 1000 realisations of the test statistic. The estimated probability and the theoretical probability match up to numerical precision. It can be observed that the probability of detection increases monotonically with increase in SNR. High probabilities of detection > 0.9 are achieved with $\text{SNR} > -10\text{dB}$. The estimated probability of false alarm is consistent to be around the true value of 0.05.

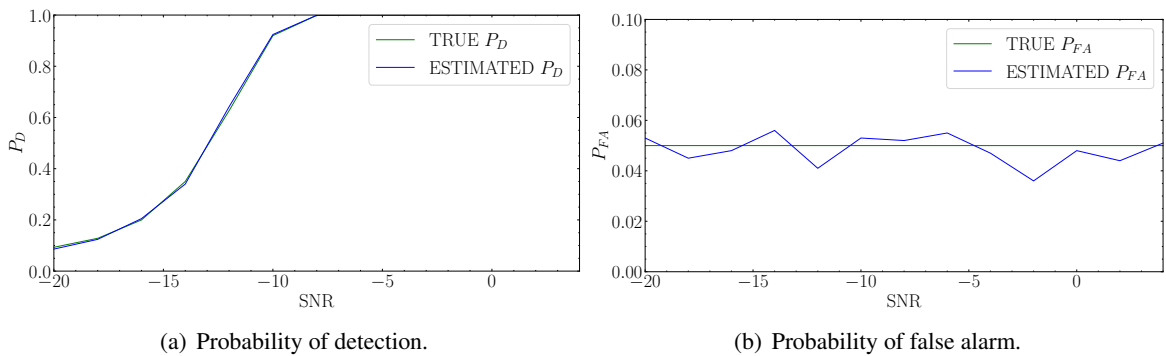


Figure 1: Monte-Carlo probabilities of the energy detector varying with SNR when the parameters are exact.

b) Monte-Carlo simulations with inexact parameters:

Figures 2(a) and 2(b) shows the variation of P_D and P_{FA} with the signal-to-noise ratio (SNR). The estimated results are averaged over 1000 realisations of the test statistic. As compared to Figures 1(a) and 1(b), the probability of detection is only as high as 0.8 at the SNR of -10dB . The probability of false alarm is also consistently higher than the true value.

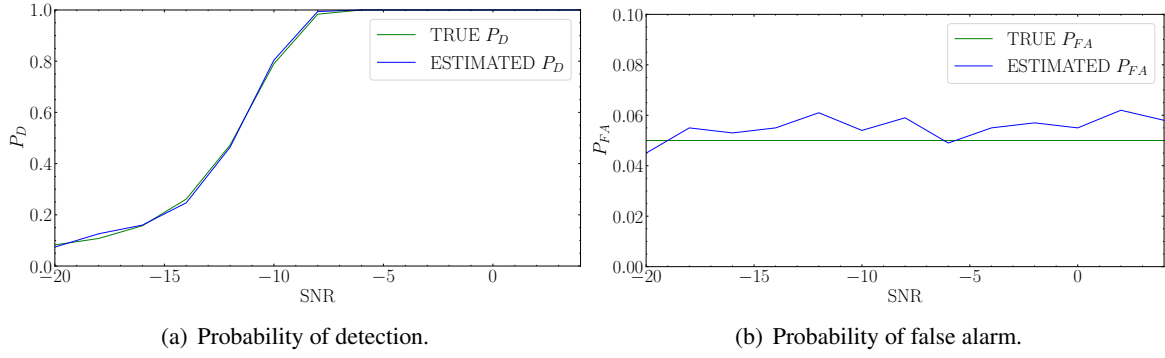


Figure 2: Monte-Carlo probabilities of the energy detector varying with SNR when the parameters are inexact.

c) *Comparison with the Bayes' detector:* The Bayes' detector, similar to the LRT, compares the ratio of the likelihoods to the ratio of the prior probabilities. The Bayes' detector decides on \mathcal{H}_1 if:

$$\begin{aligned}
 L(\mathbf{y}) &= \frac{p_Y(\mathbf{y}; \mathcal{H}_1)}{p_Y(\mathbf{y}; \mathcal{H}_0)} > \frac{P[\mathcal{H}_0]}{P[\mathcal{H}_1]}, \\
 &= \frac{\frac{1}{(2\pi(\sigma_s^2 + \sigma_w^2))^{N/2}} \exp\left(-\frac{1}{2(\sigma_s^2 + \sigma_w^2)} \sum_{n=0}^N |y[n]|^2\right)}{\frac{1}{(2\pi\sigma_w^2)^{N/2}} \exp\left(-\frac{1}{2\sigma_w^2} \sum_{n=0}^N |y[n]|^2\right)} > \frac{P[\mathcal{H}_0]}{P[\mathcal{H}_1]} \\
 &= \left(\frac{\sigma_w^2}{\sigma_s^2 + \sigma_w^2}\right)^{N/2} \exp\left(\frac{1}{2} \frac{\sigma_s^2}{\sigma_w^2 (\sigma_s^2 + \sigma_w^2)} \sum_{n=0}^{N-1} |y[n]|^2\right) > \frac{P[\mathcal{H}_0]}{P[\mathcal{H}_1]} \\
 \Rightarrow T(\mathbf{y}) &= \sum_{n=0}^{N-1} |y[n]|^2 > 2 \left(\frac{(\sigma_s^2 + \sigma_w^2) \sigma_w^2}{\sigma_s^2} \right) \left(\frac{N}{2} \ln \left(\frac{\sigma_s^2 + \sigma_w^2}{\sigma_w^2} \right) + \ln \left(\frac{P[\mathcal{H}_0]}{P[\mathcal{H}_1]} \right) \right).
 \end{aligned} \tag{18}$$

Figure 3 shows the variation of the thresholds of the Neymann-Pearson detector and the Bayes' detector with the SNR. It can be observed that the thresholds decrease monotonically with SNR as the energy in the noise decreases with increasing SNR, and a small threshold reliably detects the signal. Between the detectors, it can be observed that, for a probability of false alarm $P_{FA} = 0.05$ in the Neymann-Pearson detector and a prior $P[\mathcal{H}_0] = 0.2$ in the Bayes' detector, the thresholds are similar. The threshold of the Neymann-Pearson detector is marginally smaller than the threshold of the Bayes' detector at lower SNR, and gradually meet as the SNR increases.

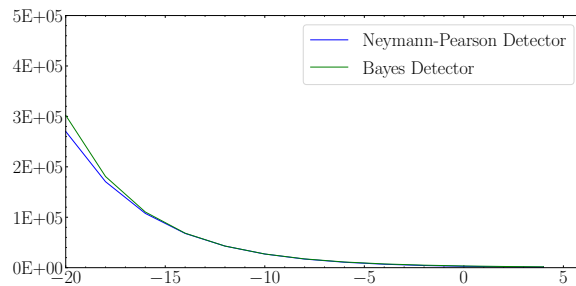


Figure 3: Variation of Neymann-Pearson threshold vs. Bayes' threshold with SNR.

Cyclostationarity Detector

a) Distribution of the test statistics:

Figures 4(a) and 4(b) shows the real and imaginary parts of the density function of the test statistic under \mathcal{H}_0 , and Figures 4(c) and 4(d) shows the real and imaginary parts of the density function of the test statistic under \mathcal{H}_1 . The histogram is plotted using 1000 realisation of each test statistic, and is compared with the Gaussian PDF with parameters derived. The approximation is known to improve with longer data samples N_d . With $N_d = 32$ and $K = 50$, the distributions are fairly well approximated by a Gaussian random variable.

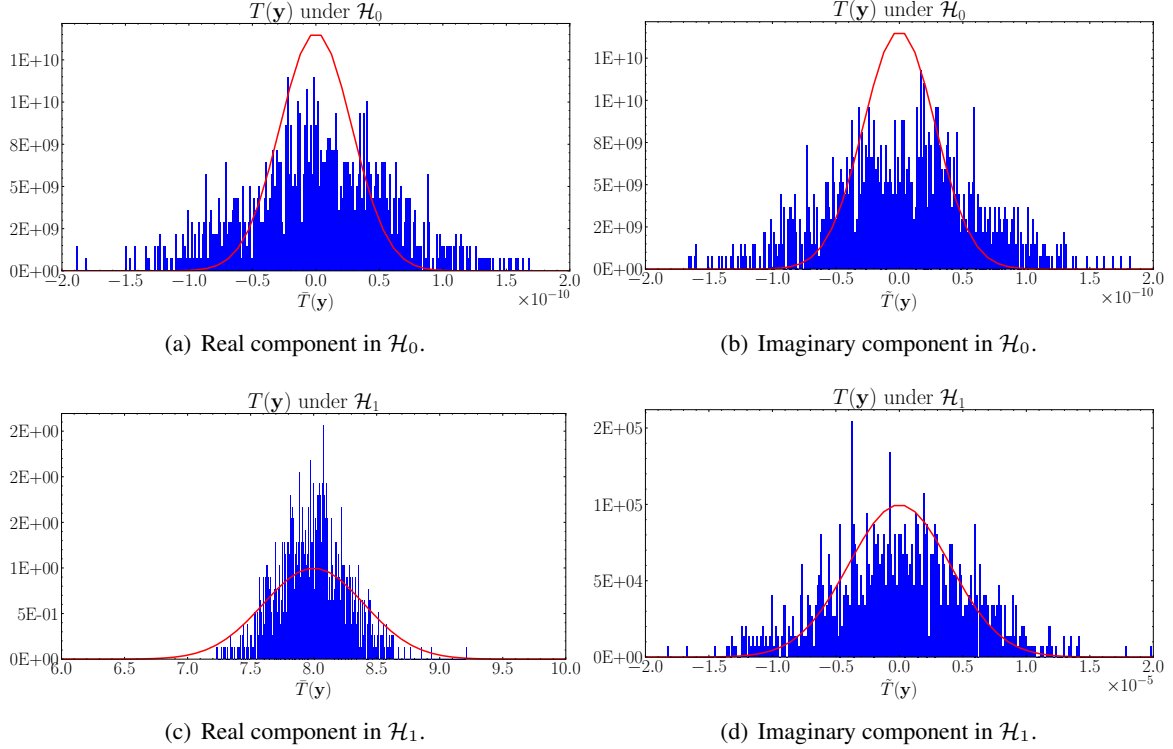


Figure 4: Monte-Carlo distributions of the test statistics of the cyclostationarity detector.

b) Monte-Carlo simulations with exact parameters:

Figures 5(a) and 5(b) shows the variation of P_D and P_{FA} with the signal-to-noise ratio (SNR). The estimated results are averaged over 1000 realisations of the test statistic. The estimated probability and the theoretical probability match up to numerical precision. It can be observed that the probability of detection increases monotonically with increase in SNR. As compared to the energy detector, the cyclostationarity detector gives probability of detection > 0.9 for SNR only > -7 dB, which is 3dB higher than the energy detector. However, the probability of false alarm is consistently, much lower than the probability of false alarm in the energy detector.

c) Monte-Carlo simulations with inexact parameters:

Figures 6(a) and 6(b) shows the variation of P_D and P_{FA} with the signal-to-noise ratio (SNR). The estimated results are averaged over 1000 realisations of the test statistic. As compared to Figures 5(a) and 5(b), the probability of detection is only as high as 0.8 at the SNR of -7 dB. The probability of false alarm is marginally higher than in Figure 5(b).

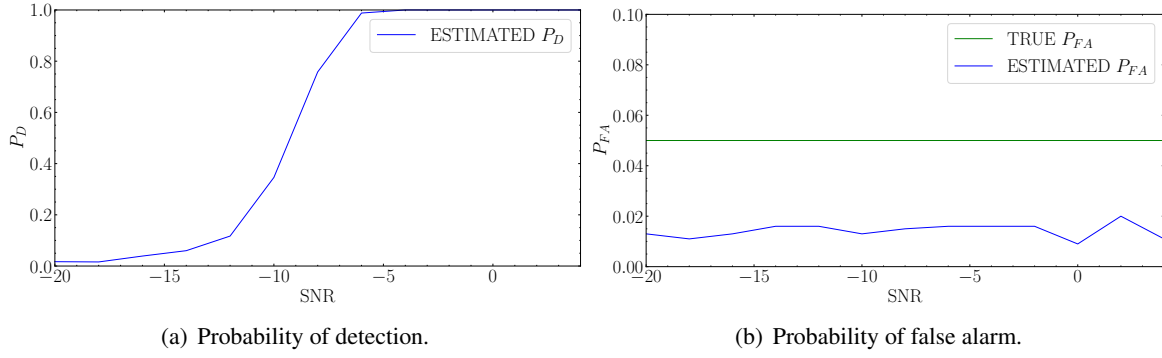


Figure 5: Monte-Carlo probabilities of the cyclostationary detector varying with SNR when the parameters are exact.

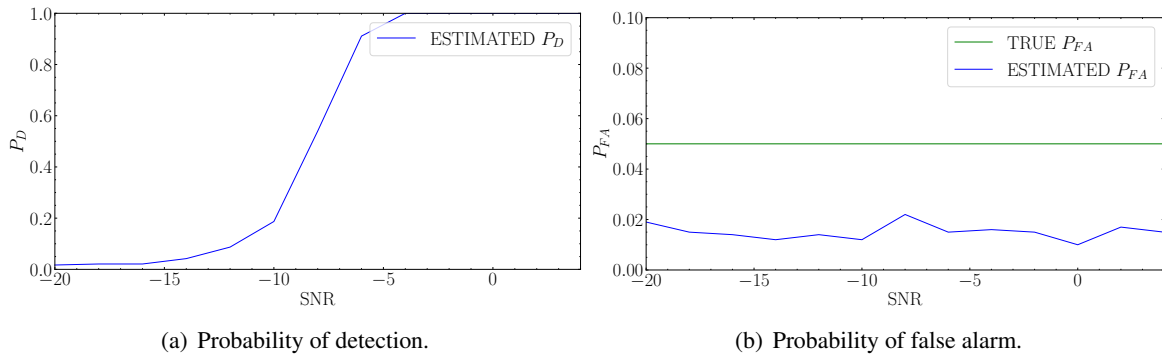


Figure 6: Monte-Carlo probabilities of the cyclostationary detector varying with SNR when the parameters are inexact.

Scripts

The Python3 scripts to generate all figures can be downloaded from the GitHub repository https://github.com/kamath-abhijith/Spectrum_Sensing. Use `requirements.txt` to install all dependencies. Also, see the following code snippets for reference.

Implementation of Energy Detector

The relevant functions are in `utils.py`.

```

1  '''
2
3  NEYMANN-PEARSON SIGNAL DETECTOR FOR
4  SPECTRUM SAMPLING IN COGNITIVE RADIO
5  BASED ON SIGNAL ENERGY
6
7  AUTHOR: ABIJITH J. KAMATH
8  abijithj@iisc.ac.in, kamath-abhijith.github.io
9
10 '''
11
12 # %% LOAD LIBRARIES
13
14 import os
15 import numpy as np
16

```

```

17 from tqdm import tqdm
18
19 from scipy.stats import chi2
20
21 from matplotlib import style
22 from matplotlib import rcParams
23 from matplotlib import pyplot as plt
24
25 import utils
26
27 # %% PLOT SETTINGS
28
29 plt.style.use(['science', 'ieee'])
30
31 plt.rcParams.update({
32     "font.family": "serif",
33     "font.serif": ["cm"],
34     "mathtext.fontset": "cm",
35     "font.size": 24})
36
37 # %% PARAMETERS
38
39 Nd = 32
40 Nc = 8
41 K = 50
42 N = (K+1)*(Nc+Nd)
43 NUM_STATS = 1000
44
45 PFA = 0.05
46
47 SNR_MIN = -20
48 SNR_MAX = 6
49 SNR_STEP = 2
50
51 # %% MONTE CARLO SIMULATIONS // CLEAN PARAMETERS
52
53 SNRS = np.arange(SNR_MIN, SNR_MAX, SNR_STEP)
54
55 true_PFA = np.zeros(len(SNRS))
56 true_PD = np.zeros(len(SNRS))
57 est_PFA = np.zeros(len(SNRS))
58 est_PD = np.zeros(len(SNRS))
59
60 for itr, SNR in tqdm(enumerate(SNRS)):
61     noise_var = 1 / 10**(SNR/10)
62     threshold = chi2.isf(q=PFA, df=N) * noise_var
63
64     stats_H0 = utils.energy_stat_H0(NUM_STATS, Nd, Nc, K, noise_var)
65     stats_H1 = utils.energy_stat_H1(NUM_STATS, Nd, Nc, K, noise_var)
66
67     false_alarms = sum(stats_H0 > threshold)
68     detections = sum(stats_H1 > threshold)
69
70     est_PFA[itr] = false_alarms / NUM_STATS
71     est_PD[itr] = detections / NUM_STATS
72
73     true_PFA[itr] = PFA
74     true_PD[itr] = chi2.sf(x=threshold / (1 + noise_var), df=N)
75
76 # %% PLOTS // CLEAN PARAMETERS
77

```



```

78 os.makedirs('./results/', exist_ok=True)
79 path = './results/'
80
81 plt.figure(figsize=(12,6))
82 ax = plt.gca()
83 utils.plot_signal(SNRS, true_PD, ax=ax, plot_colour='green',
84     legend_label=r'TRUE  $P_D$ ', show=False)
85 utils.plot_signal(SNRS, est_PD, ax=ax, plot_colour='blue',
86     legend_label=r'ESTIMATED  $P_D$ ', yaxis_label=r' $P_D$ ',
87     xaxis_label=r' $\mathrm{SNR}$ ', show=True,
88     save=path+'eneProb_PD')
89
90 plt.figure(figsize=(12,6))
91 ax = plt.gca()
92 utils.plot_signal(SNRS, true_PFA, ax=ax, plot_colour='green',
93     legend_label=r'TRUE  $P_{FA}$ ', show=False)
94 utils.plot_signal(SNRS, est_PFA, ax=ax, plot_colour='blue',
95     legend_label=r'ESTIMATED  $P_{FA}$ ', yaxis_label=r' $P_{FA}$ ',
96     xaxis_label=r' $\mathrm{SNR}$ ', ylimits=[0,2*PFA], show=True,
97     save=path+'eneProb_PFA')
98
99 # %% MONTE CARLO SIMULATIONS // NOISY PARAMETERS
100
101 SNRS = np.arange(SNR_MIN, SNR_MAX, SNR_STEP)
102
103 true_PFA = np.zeros(len(SNRS))
104 true_PD = np.zeros(len(SNRS))
105 est_PFA = np.zeros(len(SNRS))
106 est_PD = np.zeros(len(SNRS))
107
108 for itr, SNR in tqdm(enumerate(SNRS)):
109     noise_var = 1 / 10**(SNR/10)
110     noise_var = noise_var * 10**(1/10)
111     threshold = chi2.isf(q=PFA, df=N) * noise_var
112
113     stats_H0 = utils.energy_stat_H0(NUM_STATS, Nd, Nc, K, noise_var)
114     stats_H1 = utils.energy_stat_H1(NUM_STATS, Nd, Nc, K, noise_var)
115
116     false_alarms = sum(stats_H0 > threshold)
117     detections = sum(stats_H1 > threshold)
118
119     est_PFA[itr] = false_alarms / NUM_STATS
120     est_PD[itr] = detections / NUM_STATS
121
122     true_PFA[itr] = PFA
123     true_PD[itr] = chi2.sf(x=threshold / (1 + noise_var), df=N)
124
125 # %% PLOTS // NOISY PARAMETERS
126
127 os.makedirs('./results/', exist_ok=True)
128 path = './results/'
129
130 plt.figure(figsize=(12,6))
131 ax = plt.gca()
132 utils.plot_signal(SNRS, true_PD, ax=ax, plot_colour='green',
133     legend_label=r'TRUE  $P_D$ ', show=False)
134 utils.plot_signal(SNRS, est_PD, ax=ax, plot_colour='blue',
135     legend_label=r'ESTIMATED  $P_D$ ', yaxis_label=r' $P_D$ ',
136     xaxis_label=r' $\mathrm{SNR}$ ', show=True,
137     save=path+'eneProb_PD_Noisy')
138

```

```

139 plt.figure(figsize=(12,6))
140 ax = plt.gca()
141 utils.plot_signal(SNRS, true_PFA, ax=ax, plot_colour='green',
142     legend_label=r'TRUE  $P_{FA}$ ', show=False)
143 utils.plot_signal(SNRS, est_PFA, ax=ax, plot_colour='blue',
144     legend_label=r'ESTIMATED  $P_{FA}$ ', yaxis_label=r' $P_{FA}$ ',
145     xaxis_label=r' $\mathrm{SNR}$ ', ylimits=[0,2*PFA], show=True,
146     save=path+'eneProb_PFA_Noisy')
147
148 # %% THRESHOLD COMPARISONS
149
150 prior1 = 0.2
151 prior0 = 1-prior1
152 SNRS = np.arange(SNR_MIN, SNR_MAX, SNR_STEP)
153
154 threshold_NP = np.zeros(len(SNRS))
155 threshold_BD = np.zeros(len(SNRS))
156 for itr, SNR in tqdm(enumerate(SNRS)):
157     noise_var = 1 / 10**(SNR/10)
158     noise_var = noise_var * 10**(1/10)
159
160     threshold_NP[itr] = chi2.isf(q=PFA, df=N) * noise_var
161     threshold_BD[itr] = 2 * (1+noise_var) * noise_var * \
162         (N/2 * np.log((1+noise_var)/noise_var) + np.log(prior0/prior1))
163
164 # %% PLOTS :: THRESHOLD COMPARISON
165
166 os.makedirs('./results/', exist_ok=True)
167 path = './results/'
168
169 plt.figure(figsize=(12,6))
170 ax = plt.gca()
171 utils.plot_signal(SNRS, threshold_NP, ax=ax,
172     legend_label=r'Neymann-Pearson Detector', show=False)
173 utils.plot_signal(SNRS, threshold_BD, ax=ax, plot_colour='green',
174     ylimits=[0,0.5*1e6], legend_label=r'Bayes Detector', show=True,
175     save=path+'thresholds')
176 # %%

```

Implementation of Cyclostationary Detector

The relevant functions are in `utils.py`.

```

1 '''
2
3 NEYMANN-PEARSON SIGNAL DETECTOR FOR
4 SPECTRUM SAMPLING IN COGNITIVE RADIO
5 BASED ON CYCLOSTATIONARITY
6
7 AUTHOR: ABIJITH J. KAMATH
8 abijithj@iisc.ac.in, kamath-abhijith.github.io
9
10 '''
11
12 # %% LOAD LIBRARIES
13
14 import os
15 import numpy as np
16
17 from tqdm import tqdm

```

```

18
19 from scipy.stats import chi2
20
21 from matplotlib import style
22 from matplotlib import rcParams
23 from matplotlib import pyplot as plt
24
25 import utils
26
27 # %% PLOT SETTINGS
28
29 plt.style.use(['science', 'ieee'])
30
31 plt.rcParams.update({
32     "font.family": "serif",
33     "font.serif": ["cm"],
34     "mathtext.fontset": "cm",
35     "font.size": 24})
36
37 # %% PARAMETERS
38
39 Nd = 32
40 Nc = 8
41 K = 50
42 N = (K+1)*(Nc+Nd)
43 NUM_STATS = 1000
44
45 PFA = 0.05
46
47 SNR_MIN = -20
48 SNR_MAX = 6
49 SNR_STEP = 2
50
51 # %% DISTRIBUTION OF TEST STATISTIC
52
53 SNR = 100
54 noise_var = 1 / 10** (SNR/10)
55
56 stats_H0 = utils.cyclo_stat_H0(NUM_STATS, Nd, Nc, K, noise_var)
57 stats_H1 = utils.cyclo_stat_H1(NUM_STATS, Nd, Nc, K, noise_var)
58
59 # %% PLOTS :: DISTRIBUTION OF TEST STATISTIC
60
61 os.makedirs('./results/', exist_ok=True)
62 path = './results/'
63
64 plt.figure(figsize=(12,6))
65 ax = plt.gca()
66 utils.plot_gaussian(np.linspace(-2*1e-10,2*1e-10), 0, Nc/(2*K)*noise_var**2,
67     ax=ax, show=False)
68 utils.plot_histogram(np.real(stats_H0), 256, ax=ax,
69     xaxis_label=r'$\bar{T}(\mathbf{y})$',
70     title_text=r'$T(\mathbf{y})$ under $\mathcal{H}_0$', show=True, save=path+'
71     cycDist_H0R')
72
73 plt.figure(figsize=(12,6))
74 ax = plt.gca()
75 utils.plot_gaussian(np.linspace(-2*1e-10,2*1e-10), 0, Nc/(2*K)*noise_var**2,
76     ax=ax, show=False)
77 utils.plot_histogram(np.imag(stats_H0), 256, ax=ax,
78     xaxis_label=r'$\tilde{T}(\mathbf{y})$',

```

```

78     title_text=r'$T(\mathbf{y})$ under $\mathcal{H}_0$', show=True, save=path+'
    cycDist_H0I')
79
80 plt.figure(figsize=(12,6))
81 ax = plt.gca()
82 utils.plot_gaussian(np.linspace(6,10), Nc, Nc/K*(1+noise_var+(noise_var**2)/2),
83     ax=ax, show=False)
84 utils.plot_histogram(np.real(stats_H1), 256, ax=ax,
85     xaxis_label=r'$\tilde{T}(\mathbf{y})$', xlims=[6,10],
86     title_text=r'$T(\mathbf{y})$ under $\mathcal{H}_1$', show=True, save=path+'
    cycDist_H1R')
87
88 plt.figure(figsize=(12,6))
89 ax = plt.gca()
90 utils.plot_gaussian(np.linspace(-2*1e-5,2*1e-5), 0, Nc/K*(noise_var+(noise_var**2)/2),
91     ax=ax, show=False)
92 utils.plot_histogram(np.imag(stats_H1), 256, ax=ax,
93     xaxis_label=r'$\tilde{T}(\mathbf{y})$', xlims=[-2*1e-5,2*1e-5],
94     title_text=r'$T(\mathbf{y})$ under $\mathcal{H}_1$', show=True, save=path+'
    cycDist_H1I')
95
96 # %% MONTE CARLO SIMULATIONS // CLEAN PARAMETERS
97
98 SNRS = np.arange(SNR_MIN, SNR_MAX, SNR_STEP)
99
100 true_PFA = np.zeros(len(SNRS))
101 est_PFA = np.zeros(len(SNRS))
102 est_PD = np.zeros(len(SNRS))
103
104 for itr, SNR in tqdm(enumerate(SNRS)):
105     noise_var = 1 / 10**((SNR/10))
106     threshold = chi2.isf(q=PFA, df=2) * Nc / K * (noise_var**2)
107
108     stats_H0 = utils.cyclo_stat_H0(NUM_STATS, Nd, Nc, K, noise_var)
109     stats_H1 = utils.cyclo_stat_H1(NUM_STATS, Nd, Nc, K, noise_var)
110
111     false_alarms = sum(np.square(np.abs(stats_H0)) > threshold)
112     detections = sum(np.square(np.abs(stats_H1)) > threshold)
113
114     est_PFA[itr] = false_alarms / NUM_STATS
115     est_PD[itr] = detections / NUM_STATS
116
117     true_PFA[itr] = PFA
118
119 # %% PLOTS :: MONTE CARLO SIMULATIONS // CLEAN PARAMETERS
120
121 plt.figure(figsize=(12,6))
122 ax = plt.gca()
123 utils.plot_signal(SNRS, est_PD, ax=ax, plot_colour='blue',
124     legend_label=r'ESTIMATED $P_{D}$', yaxis_label=r'$P_{D}$',
125     xaxis_label=r'$\mathrm{SNR}$', show=True, save=path+'cycProb_PD')
126
127 plt.figure(figsize=(12,6))
128 ax = plt.gca()
129 utils.plot_signal(SNRS, true_PFA, ax=ax, plot_colour='green',
130     legend_label=r'TRUE $P_{FA}$', show=False)
131 utils.plot_signal(SNRS, est_PFA, ax=ax, plot_colour='blue',
132     legend_label=r'ESTIMATED $P_{FA}$', yaxis_label=r'$P_{FA}$',
133     xaxis_label=r'$\mathrm{SNR}$', ylims=[0,2*PFA], show=True,
134     save=path+'cycProb_PFA')
135

```

```

136 # %% MONTE CARLO SIMULATIONS // NOISY PARAMETERS
137
138 SNRS = np.arange(SNR_MIN, SNR_MAX, SNR_STEP)
139
140 true_PFA = np.zeros(len(SNRS))
141 est_PFA = np.zeros(len(SNRS))
142 est_PD = np.zeros(len(SNRS))
143
144 for itr, SNR in tqdm(enumerate(SNRS)):
145     noise_var = 1 / 10**(SNR/10)
146     noise_var = noise_var * 10**(1/10)
147     threshold = chi2.isf(q=PFA, df=2) * Nc / K * (noise_var**2)
148
149     stats_H0 = utils.cyclo_stat_H0(NUM_STATS, Nd, Nc, K, noise_var)
150     stats_H1 = utils.cyclo_stat_H1(NUM_STATS, Nd, Nc, K, noise_var)
151
152     false_alarms = sum(np.square(np.abs(stats_H0)) > threshold)
153     detections = sum(np.square(np.abs(stats_H1)) > threshold)
154
155     est_PFA[itr] = false_alarms / NUM_STATS
156     est_PD[itr] = detections / NUM_STATS
157
158     true_PFA[itr] = PFA
159
160 # %% PLOTS :: MONTE CARLO SIMULATIONS // NOISY PARAMETERS
161
162 plt.figure(figsize=(12,6))
163 ax = plt.gca()
164 utils.plot_signal(SNRS, est_PD, ax=ax, plot_colour='blue',
165     legend_label=r'ESTIMATED  $P_D$ ', yaxis_label=r' $P_D$ ',
166     xaxis_label=r' $\mathrm{SNR}$ ', show=True, save=path+'cycProb_PD_Noisy')
167
168 plt.figure(figsize=(12,6))
169 ax = plt.gca()
170 utils.plot_signal(SNRS, true_PFA, ax=ax, plot_colour='green',
171     legend_label=r'TRUE  $P_{FA}$ ', show=False)
172 utils.plot_signal(SNRS, est_PFA, ax=ax, plot_colour='blue',
173     legend_label=r'ESTIMATED  $P_{FA}$ ', yaxis_label=r' $P_{FA}$ ',
174     xaxis_label=r' $\mathrm{SNR}$ ', ylimits=[0,2*PFA], show=True,
175     save=path+'cycProb_PFA_Noisy')

```

utils.py

This script contains all the relevant functions and helpers.

```

1 '''
2
3 TOOLS FOR SPECTRUM SENSING FOR COGNITIVE RADIO
4
5 AUTHOR: ABIJITH J. KAMATH
6 abijithj@iisc.ac.in, kamath-abhijith.github.io
7
8 '''
9
10 # %% LOAD LIBRARIES
11
12 import numpy as np
13
14 from matplotlib import pyplot as plt
15 from matplotlib.ticker import StrMethodFormatter

```

```

16
17 from scipy.stats import multivariate_normal
18
19 # %% PLOTTING FUNCTIONS
20
21 def plot_signal(x, y, ax=None, plot_colour='blue', xaxis_label=None,
22               yaxis_label=None, title_text=None, legend_label=None, legend_show=True,
23               legend_loc='upper right', line_style='-', line_width=None,
24               show=False, xlims=[-20,6], ylims=[0,1], save=None):
25     '''
26     Plots signal with abscissa in x and ordinates in y
27
28     '''
29     if ax is None:
30         fig = plt.figure(figsize=(12,6))
31         ax = plt.gca()
32
33     plt.plot(x, y, linestyle=line_style, linewidth=line_width, color=plot_colour,
34             label=legend_label)
35     if legend_label and legend_show:
36         plt.legend(loc=legend_loc, frameon=True, framealpha=0.8, facecolor='white')
37     plt.xlabel(xaxis_label)
38     plt.ylabel(yaxis_label)
39
40     plt.xlim(xlims)
41     plt.ylim(ylims)
42     plt.title(title_text)
43
44     plt.gca().yaxis.set_major_formatter(StrMethodFormatter('{x:,.0E}'))
45
46     if save:
47         plt.savefig(save + '.pdf', format='pdf')
48
49     if show:
50         plt.show()
51
52     return
53
54 def plot_histogram(x, num_bins, ax=None, plot_colour='blue', xaxis_label=None,
55                  yaxis_label=None, title_text=None, xlims=[-2*1e-10,2*1e-10], show=False, save=None)
56 :
57     '''
58     Plots histogram of data in x
59
60     '''
61
62     if ax is None:
63         fig = plt.figure(figsize=(12,6))
64         ax = plt.gca()
65
66     plt.hist(x, bins=num_bins, color=plot_colour, density=True)
67
68     plt.xlabel(xaxis_label)
69     plt.ylabel(yaxis_label)
70     plt.title(title_text)
71
72     plt.xlim(xlims)
73     # plt.ylim([0,30])
74
75     plt.gca().yaxis.set_major_formatter(StrMethodFormatter('{x:,.0E}'))

```

```

76     if save:
77         plt.savefig(save + '.pdf', format='pdf')
78
79     if show:
80         plt.show()
81
82     return
83
84 def plot_gaussian(xvals, mean, var, ax=None, plot_colour='red',
85                  xaxis_label=None, yaxis_label=None, line_style='-', line_width=2,
86                  title_text=None, show=False, save=None):
87     '''
88     Plots 1D Gaussian with mean and variance
89
90     '''
91
92     if ax is None:
93         fig = plt.figure(figsize=(12,6))
94         ax = plt.gca()
95
96     gaussian = multivariate_normal.pdf(xvals, mean, var)
97     plt.plot(xvals, gaussian, linestyle=line_style, linewidth=line_width,
98             color=plot_colour)
99
100    plt.xlabel(xaxis_label)
101    plt.ylabel(yaxis_label)
102    plt.title(title_text)
103
104    plt.gca().yaxis.set_major_formatter(StrMethodFormatter('{x:,.0E}'))
105
106    if save:
107        plt.savefig(save + '.pdf', format='pdf')
108
109    if show:
110        plt.show()
111
112    return
113
114 # %% SIGNALS
115
116 def generate_QPSK_block(Nd, Nc):
117     '''
118     Generates a QPSK block with unit variance
119
120     :param Nd: IFFT length
121     :param Nc: length of cyclic repetition
122
123     :return: OFDM signal block
124
125     '''
126
127     s = np.zeros(Nd)
128     for k in range(Nd):
129         s[k] = (2*np.random.binomial(1, 0.5)-1)/np.sqrt(2) + \
130             1j * (2*np.random.binomial(1, 0.5)-1)/np.sqrt(2)
131
132     x = np.fft.ifft(s) * np.sqrt(2*Nd)
133
134     return np.hstack([x[:Nc], x])
135
136 def generate_QPSK_signal(Nd, Nc, K):

```

```

137     '''
138     Generate a QPSK signal with unit variance
139
140     :param Nd: IFFT length of each block
141     :param Nc: length of cyclic repetition
142     :param K: number of OFDM blocks
143
144     :return: K+1 OFDM signal blocks
145
146     '''
147
148     signal = generate_QPSK_block(Nd, Nc)
149     for _ in range(K):
150         signal = np.hstack([signal, generate_QPSK_block(Nd, Nc)])
151
152     return signal
153
154 # %% DETECTORS
155
156 def energy_stat_H0(num_stats, Nd, Nc, K, noise_var):
157     '''
158     Generates signal energy statistics for noise only hypothesis
159
160     :param num_stats: number of realisations
161     :param Nd: length of IFFT
162     :param Nc: length of cyclic repetitions
163     :param K: number of OFDM blocks
164     :param noise_var: variance of AWGN channel
165
166     :return: energy statistics
167
168     '''
169
170     stats = np.zeros(num_stats)
171     for itr in range(num_stats):
172         noise = np.sqrt(noise_var)*np.random.randn((K+1)*(Nd+Nc))
173
174         stats[itr] = np.sum(np.square(np.abs(noise)))
175
176     return stats
177
178 def energy_stat_H1(num_stats, Nd, Nc, K, noise_var):
179     '''
180     Generates signal energy statistics for signal present hypothesis
181
182     :param num_stats: number of realisations
183     :param Nd: length of IFFT
184     :param Nc: length of cyclic repetitions
185     :param K: number of OFDM blocks
186     :param noise_var: variance of AWGN channel
187
188     :return: energy statistics
189
190     '''
191
192     stats = np.zeros(num_stats)
193     for itr in range(num_stats):
194         signal = generate_QPSK_signal(Nd, Nc, K)
195         noise = np.sqrt(noise_var)*np.random.randn((K+1)*(Nd+Nc))
196
197         stats[itr] = np.sum(np.square(np.abs(signal + noise)))

```



```

198
199     return stats
200
201 def cyclo_stat_H0(num_stats, Nd, Nc, K, noise_var):
202     '''
203     Generates cyclostationary ACF statistic for noise only hypothesis
204
205     :param num_stats: number of realisations
206     :param Nd: length of IFFT
207     :param Nc: length of cyclic repetitions
208     :param K: number of OFDM blocks
209     :param noise_var: variance of AWGN channel
210
211     :return: cyclostationary ACF
212
213     '''
214
215     stats = np.zeros(num_stats, dtype=np.complex)
216     for itr in range(num_stats):
217         noise = np.sqrt(noise_var)*np.random.randn((K+1)*(Nd+Nc),2).view(np.complex)
218         y = noise
219
220         stat = np.complex(0)
221         for n in range(Nc):
222             for k in range(K):
223                 stat += y[n+k*(Nc+Nd)] * np.conjugate(y[n+k*(Nc+Nd)+Nd])
224
225         stats[itr] = stat/K
226
227     return stats
228
229 def cyclo_stat_H1(num_stats, Nd, Nc, K, noise_var):
230     '''
231     Generates cyclostationary ACF statistic for signal present hypothesis
232
233     :param num_stats: number of realisations
234     :param Nd: length of IFFT
235     :param Nc: length of cyclic repetitions
236     :param K: number of OFDM blocks
237     :param noise_var: variance of AWGN channel
238
239     :return: cyclostationary ACF
240
241     '''
242
243     stats = np.zeros(num_stats, dtype=np.complex)
244     for itr in range(num_stats):
245         signal = generate_QPSK_signal(Nd, Nc, K)[: ,None]
246         for k in range(K):
247             signal[k*(Nc+Nd):k*(Nc+Nd)+Nc] = signal[k*(Nc+Nd)+Nd:(k+1)*(Nc+Nd)]
248         noise = np.sqrt(noise_var)*np.random.randn((K+1)*(Nd+Nc),2).view(np.complex)
249         y = signal + noise
250
251         stat = np.complex(0)
252         for n in range(Nc):
253             for k in range(K):
254                 stat += y[n+k*(Nc+Nd)] * np.conjugate(y[n+k*(Nc+Nd)+Nd])
255
256         stats[itr] = stat/K
257
258     return stats

```