




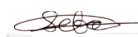
UNIVERSITEIT VAN PRETORIA  
UNIVERSITY OF PRETORIA  
YUNIBESITHI YA PRETORIA

Faculty of Engineering, Built Environment and  
Information Technology

## EDC 310

### DIGITAL COMMUNICATION

#### PRACTICAL 2: DFE AND MLSE

Name and Surname	Student Number	Signature	% Contribution
Kwaku Bediako	u04465483		30
Lefa Raleting	u14222460		70

By signing this assignment we confirm that we have read and are aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

October 10, 2020

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theoretical background</b>	<b>2</b>
2.1	DFE . . . . .	2
2.2	MLSE . . . . .	2
<b>3</b>	<b>Design/Method</b>	<b>3</b>
3.1	DFE . . . . .	3
3.2	MLSE . . . . .	5
<b>4</b>	<b>Simulations and Results</b>	<b>5</b>
4.1	DFE . . . . .	5
4.2	MLSE . . . . .	6
<b>5</b>	<b>Discussion</b>	<b>8</b>
<b>6</b>	<b>Conclusion</b>	<b>9</b>
<b>7</b>	<b>References</b>	<b>9</b>
<b>8</b>	<b>Appendix A</b>	<b>10</b>
8.1	Code . . . . .	10
8.1.1	MLSE: Static CIR . . . . .	10
8.1.2	DFE: Static CIR . . . . .	19
8.1.3	MLSE: Dynamic CIR . . . . .	27
8.1.4	DFE Dynamic CIR . . . . .	37

# 1 Introduction

In the ideal case radio waves travel in a straight line (**Line of sight**) between the transmitter and receiver, meaning all transmitted signals arrive at the same time, however that's not always the case in the practical world, there are object/obstacles, with different material compositions that have different characteristics.

When the radio waves come in contact with these objects they exhibit different behaviours namely, **reflection, refraction, absorption, scattering and diffraction**. This leads to an inherent characteristics of communication systems namely **Delay spread**. Delay spread is a result of a received signal that is dispersed in time, where each transmitted signal has its own path and every path has its own delay. This leads to inter-symbol interference.

The purpose of this paper is to establish the effect of multi-path channel in the received signal, where the Channel Impulse Response(**CIR**) is of length 3 ( $L=3$ ). It distinguishes between the effect that a static and dynamic CIR has on the demodulation of the signal on the receiver end.

Two algorithms that are of interest when it comes to reversing the effect of multi-path in communication system are, Decision feedback equaliser(**DFE**) and Maximum likelihood sequence estimation(**MLSE**), which is discussed in the theoretical background. These algorithms will be tested on three modulation schemes namely **BPSK**(Binary Phase shift keying), **4-QAM**(Quadrature amplitude modulation) and **8PSK**(eight phase shift keying).

The experiment will be conducted with transmitted symbols being limited to data blocks of length 200( $N=200$ ). The Bit Error Rates(**BER**) will be compared to give a fair analysis and comparison between the different algorithms on the respective modulation schemes.

## 2 Theoretical background

### 2.1 DFE

DFE is a non linear equaliser, instead of using a simple scalar multiple of the received bits for detection, it uses past symbols(Memory) to better estimate signals that are passing through a channel. DFE reduces the interference of past symbols on the current symbol being detected on the receiver end.

For detection the DFE selects the estimate with the lowest cost and this cost is calculated as follows:

$$cost(t) = \left| r_t - \sum_{n=0}^{L-1} s_{t-n} * c_n \right|^2 \quad (1)$$

Where:

- $r_t$  - Received bit at position t
- $s_{t-n}$  - Transmitted bit at positing t-n
- $c_n$  - Channel Impulse Response (CIR)
- L - Length of the CIR list

DFE is a sub-optimal algorithm, because it only works if the the channel response the symbols pass is of a minimum phase, this entails that the channel delay spread ratio to the power delay profile is decreasing.

### 2.2 MLSE

Maximum likelihood sequence estimation (**MLSE**) is a mathematical algorithm to extract useful data out of a noisy data stream. MLSE makes estimations on what a stream of data was, and tries to minimise the number of incorrect symbols

MLSE determines the whole transmitted symbol block at the receiver output , to determine which symbol has the highest probability to have been transmitted, by observing the entire received signal by the receiver end.

What gives MLSE an advantage is the nonlinear processing, this advantage can properly be realised through applying the viterbi algorithm . MLSE performance is measure by the rate of correct estimations. To increase this performance , it maximizes the probability that the correct sequence of symbols is found and decreasing the total path cost.

MSLE is a optimal algorithm, it will always provide the highest sequence estimation probaility even through a dynamic CIR.

## 3 Design/Method

### 3.1 DFE

To simulate communication between a transmitter and receiver of a multi-path environment, bits are generated randomly using a uniform number generator rated and mapped to symbols based on their respective modulation schemes constellation maps.

To simulate the noise that's normally added to the signal during transmission, additive white Gaussian noise (**AGWN**) is used to simulate this. The following formula is used to emulate this:

$$r_t = [\sum_{n=0}^{L-1} s_{t-n} * c_n] + \sigma * (\alpha + j\alpha) \quad (2)$$

Where:

- $\sigma = \frac{1}{\sqrt{10^{\frac{SNR}{10}} * 2 * \log_2(M)}}$  and M is the number of symbols in the respective constellation map.
- $\alpha$  is a sample from a zero mean Gaussian distribution
- L is the length of the CIR

To detect the transmitted symbol, DFE algorithm is used. The code can be found bellow:

```
1 # -----
2 #                               Sigma calculation
3 # -----
4
5 def sigma(domain,M):# M is the number of symbols
6     sigma=[]
7     for i in domain:
8         sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M
9     )))
10    return sigma
11
12 # -----
13 #                               create noise
14 # -----
15
16 def noise(size,sigma):
17     noiseList = np.random.normal(0,sigma,size)
18     return noiseList
19
20 # -----
21 #                               add noise
```

```

22 #
-----
23
24 def addnoise(transmitted,channels,L,m,snr):# assuming transmitted comes
    with the memory symbols padded
25     recieved=[]
26     M=m
27     k=snr
28
29     sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
30     #print(sigma_)
31
32
33     for i in range(L-1,len(transmitted)):
34         sample=np.random.normal(0,1,1)[0]
35         recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*
36             channels[0+1]
37             +transmitted[i-2]*channels[2]+sigma_*(sample+(
38                 sample)*1j))
39     return recieved #without the padding
40
41 #
-----
42
43 # DFE function
44 #
-----
45
46 def DFE(recieved,channels,L,options,memory):
47     Options =options# [1,-1]# these are the option available for bpsk
48     symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
49     s=0 # symbol mover
50     for i in range(0,len(recieved)):
51         guess=[]
52         n=len(channels)-1# length of the chanel L-1
53         #calculating the product but from second position
54         sumof=0
55         for j in range(1,n):
56             sumof+= symbols[n-1+s]*channels[j]
57             n-=1
58
59         for k in Options:
60             #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols
61             [1]*channels[1]+symbols[0]*channels[2]))**2)
62             guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))
63             **2)
64         estimate=Options[guess.index(min(guess))]
65         symbols.append(estimate)
66         s+=1
67     return symbols[L-1:] #final
68 #
-----

```

### 3.2 MLSE

MLSE was designed for BPSK, 4QAM and 8PSK transmission. The deltas between from each symbol, to every other symbol was calculated and then starting at the appended symbols, the most likely bit was selected until the prepended symbols were reached.

The algorithms used to estimate the transmitted bits can be found in Appendix A

## 4 Simulations and Results

### 4.1 DFE

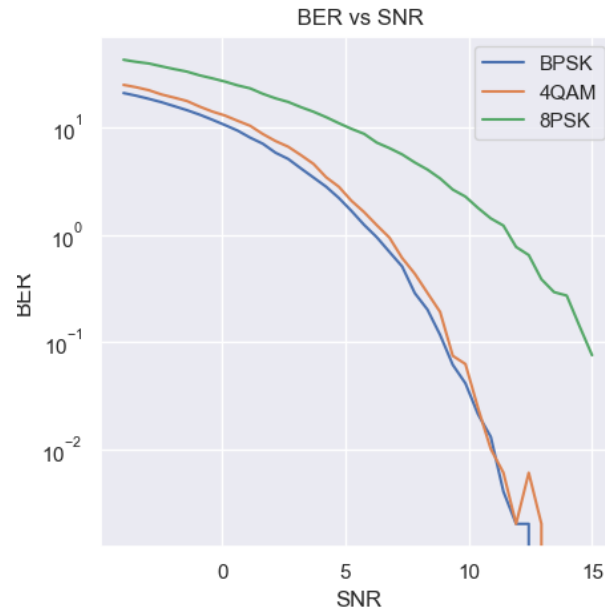


Figure 1: BER vs SNR : Static CIR

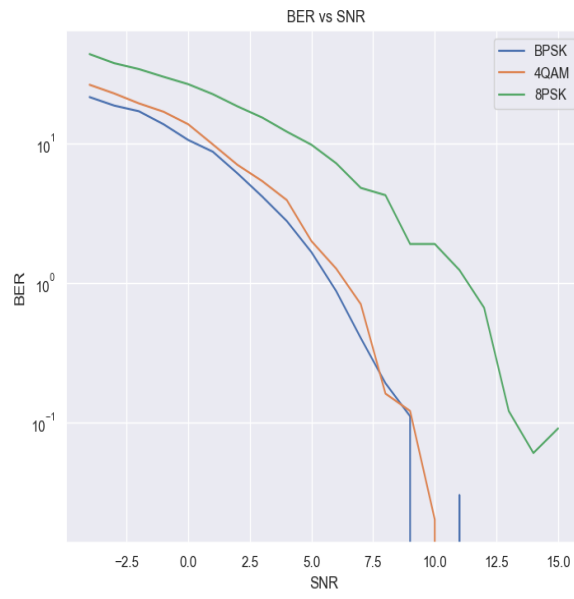


Figure 2: BER vs SNR : Dynamic CIR

## 4.2 MLSE

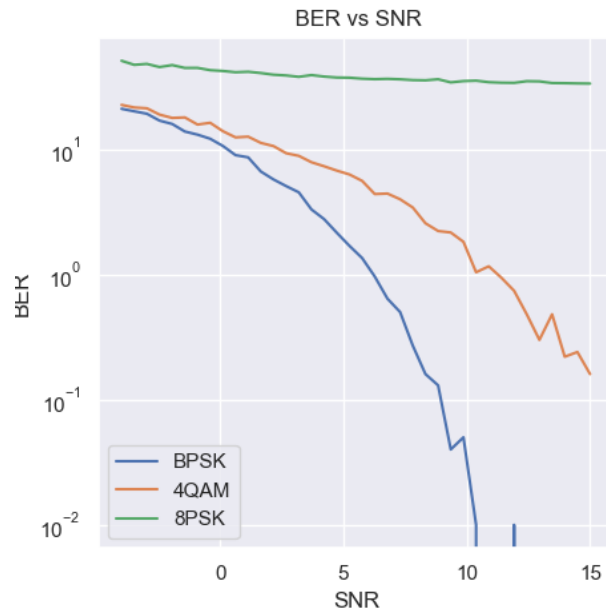


Figure 3: BER vs SNR : Static CIR



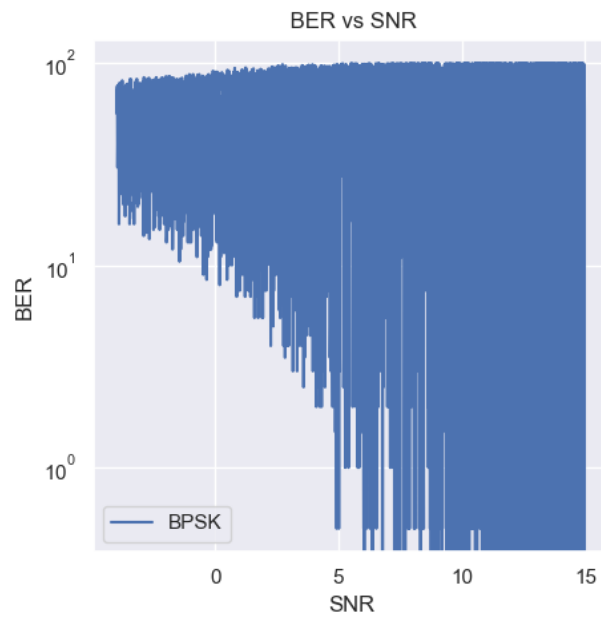


Figure 4: BER vs SNR : Dynamic CIR BPSK

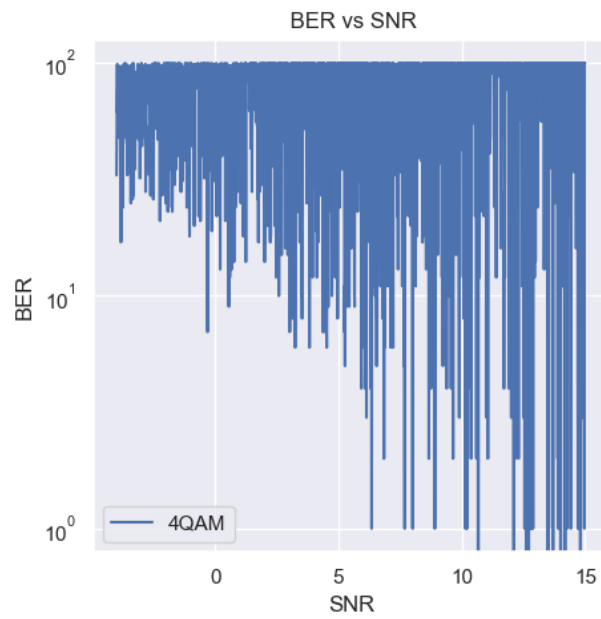


Figure 5: BER vs SNR : Dynamic CIR 4QAM

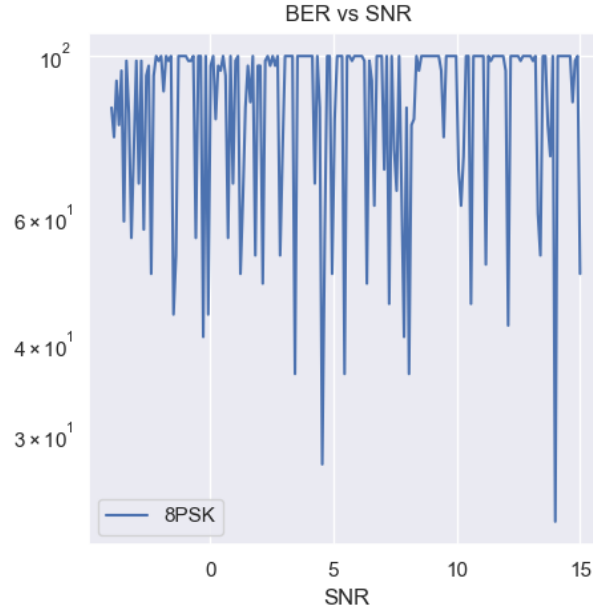


Figure 6: BER vs SNR : Dynamic CIR 8PSK

## 5 Discussion

DFE is a sub-optimal algorithm, because it only works if the the channel response the symbols pass is of a minimum phase, this entails that the channel delay spread ratio to the power delay profile is decreasing. Figure 2 shows the DFE BER vs SNR(Signal to noise Ration) for a dynamic CIR, the graph decays less smoother and a bit later.

If we compare compare it to the DFE BER vs SNR for a static CIR, it can be seen clearly that on a dynamic channel impulse response DFE performs much worse than it does on a static channel. However DFE is is fast because it needs not to check all possible paths, DFE performs much better with a decaying CIR.

It was found that with MLSE, that the bit error rate decreases more quickly for BPSL than when compared to DFE. Unexpectedly 4QAM was about as accurate for MLSE as it was for DFE, and 8PSK had a steady drop in error rate, but at a much slower rate than in DFE, remaining above 10 for all SNR values tested. As expected, BPSK continued the trend of having the lowest BER at all SNRs followed by 4QAM and then 8PSK.

With the dynamic channel, we see constant oscillation between a bit error rate of 100, and a chaninging minimum. This happens as a result of the channels switching after each block. Looking at the minimum BER, we can see faster drops in BPSK than in 4QAM, or 8PSK, but for SNR's of over 5, we register a BER of less than 1%. The BER for 8PSK also follows these general trends, however it never reaches a BER of less than 10%.

## 6 Conclusion

From the results, it was seen that DFE runs quicker than MLSE, which was expected, but surprisingly DFE gave a better more accurate results FOR static CIR. MLSE was more accurate when dealing with dynamic channels. Due to the fact that it is rare, for two wireless communicating devices to remain perfectly still through the entire transmission of data, we would expect MLSE to have superior practical use. The slower demodulation speed of MLSE is also not too concerning when one considers the current speed of processors, versus wireless transmission rates. Ways to improve Signal Accuracy, is to use techniques to boost SNR. These include use more power signals, but also allows for methods such as noise filtering.

## 7 References

- [1] B. Wichmann and D. Hill, "Building a random number generator", Byte, pp 127-128, March 1987.
- [2] G. Marsaglia and T.A. Bray, "A convenient method for generating normal variables", SIAM Rev., Vol. 6, pp 260-264, 1964.
- [3] A.Grami,"Baseband digital transmission, "Introduction to digital communication", (2016), pp 257-293

## 8 Appendix A

### 8.1 Code

#### 8.1.1 MLSE: Static CIR

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Oct  1 08:28:37 2020
4
5 @author: Kwaku
6 """
7 import random
8 import math
9 import statistics as st
10 import numpy as np
11 import matplotlib.pyplot as plt
12 from scipy.stats import norm
13 import seaborn as sns
14
15
16
17
18 # settings for seaborn plotting style
19 sns.set(color_codes=True)
20 # settings for seaborn plot sizes
21 sns.set(rc={'figure.figsize':(5,5)})
22 np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
23 #np.random.seed()
24 def theorwhichman(size):
25     rand=[]
26     for i in range(0,size):
27         rand.append(random.uniform(0, 1))
28     return rand
29 size=100
30
31 a=theorwhichman(size)
32
33 mu = st.mean(a)
34 sigma = st.stdev(a)
35
36 x = np.linspace(-1, 1, size)
37 a.sort()
38
39 print("Sigma:", sigma)
40 print("Mu:", mu)
41
42
43 #Creating bits to be transmitted
44 #
45 #
46 #
47 #
48 #
49 #
50 #
51 #
52 #
53 #
54 #
55 #
56 #
57 #
58 #
59 #
60 #
61 #
62 #
63 #
64 #
65 #
66 #
67 #
68 #
69 #
70 #
71 #
72 #
73 #
74 #
75 #
76 #
77 #
78 #
79 #
80 #
81 #
82 #
83 #
84 #
85 #
86 #
87 #
88 #
89 #
90 #
91 #
92 #
93 #
94 #
95 #
96 #
97 #
98 #
99 #
100 #
101 #
102 #
103 #
104 #
105 #
106 #
107 #
108 #
109 #
110 #
111 #
112 #
113 #
114 #
115 #
116 #
117 #
118 #
119 #
120 #
121 #
122 #
123 #
124 #
125 #
126 #
127 #
128 #
129 #
130 #
131 #
132 #
133 #
134 #
135 #
136 #
137 #
138 #
139 #
140 #
141 #
142 #
143 #
144 #
145 #
146 #
147 #
148 #
149 #
150 #
151 #
152 #
153 #
154 #
155 #
156 #
157 #
158 #
159 #
160 #
161 #
162 #
163 #
164 #
165 #
166 #
167 #
168 #
169 #
170 #
171 #
172 #
173 #
174 #
175 #
176 #
177 #
178 #
179 #
180 #
181 #
182 #
183 #
184 #
185 #
186 #
187 #
188 #
189 #
190 #
191 #
192 #
193 #
194 #
195 #
196 #
197 #
198 #
199 #
200 #
201 #
202 #
203 #
204 #
205 #
206 #
207 #
208 #
209 #
210 #
211 #
212 #
213 #
214 #
215 #
216 #
217 #
218 #
219 #
220 #
221 #
222 #
223 #
224 #
225 #
226 #
227 #
228 #
229 #
230 #
231 #
232 #
233 #
234 #
235 #
236 #
237 #
238 #
239 #
240 #
241 #
242 #
243 #
244 #
245 #
246 #
247 #
248 #
249 #
250 #
251 #
252 #
253 #
254 #
255 #
256 #
257 #
258 #
259 #
260 #
261 #
262 #
263 #
264 #
265 #
266 #
267 #
268 #
269 #
270 #
271 #
272 #
273 #
274 #
275 #
276 #
277 #
278 #
279 #
280 #
281 #
282 #
283 #
284 #
285 #
286 #
287 #
288 #
289 #
290 #
291 #
292 #
293 #
294 #
295 #
296 #
297 #
298 #
299 #
300 #
301 #
302 #
303 #
304 #
305 #
306 #
307 #
308 #
309 #
310 #
311 #
312 #
313 #
314 #
315 #
316 #
317 #
318 #
319 #
320 #
321 #
322 #
323 #
324 #
325 #
326 #
327 #
328 #
329 #
330 #
331 #
332 #
333 #
334 #
335 #
336 #
337 #
338 #
339 #
340 #
341 #
342 #
343 #
344 #
345 #
346 #
347 #
348 #
349 #
350 #
351 #
352 #
353 #
354 #
355 #
356 #
357 #
358 #
359 #
360 #
361 #
362 #
363 #
364 #
365 #
366 #
367 #
368 #
369 #
370 #
371 #
372 #
373 #
374 #
375 #
376 #
377 #
378 #
379 #
380 #
381 #
382 #
383 #
384 #
385 #
386 #
387 #
388 #
389 #
390 #
391 #
392 #
393 #
394 #
395 #
396 #
397 #
398 #
399 #
400 #
401 #
402 #
403 #
404 #
405 #
406 #
407 #
408 #
409 #
410 #
411 #
412 #
413 #
414 #
415 #
416 #
417 #
418 #
419 #
420 #
421 #
422 #
423 #
424 #
425 #
426 #
427 #
428 #
429 #
430 #
431 #
432 #
433 #
434 #
435 #
436 #
437 #
438 #
439 #
440 #
441 #
442 #
443 #
444 #
445 #
446 #
447 #
448 #
449 #
450 #
451 #
452 #
453 #
454 #
455 #
456 #
457 #
458 #
459 #
460 #
461 #
462 #
463 #
464 #
465 #
466 #
467 #
468 #
469 #
470 #
471 #
472 #
473 #
474 #
475 #
476 #
477 #
478 #
479 #
480 #
481 #
482 #
483 #
484 #
485 #
486 #
487 #
488 #
489 #
490 #
491 #
492 #
493 #
494 #
495 #
496 #
497 #
498 #
499 #
500 #
501 #
502 #
503 #
504 #
505 #
506 #
507 #
508 #
509 #
510 #
511 #
512 #
513 #
514 #
515 #
516 #
517 #
518 #
519 #
520 #
521 #
522 #
523 #
524 #
525 #
526 #
527 #
528 #
529 #
530 #
531 #
532 #
533 #
534 #
535 #
536 #
537 #
538 #
539 #
540 #
541 #
542 #
543 #
544 #
545 #
546 #
547 #
548 #
549 #
550 #
551 #
552 #
553 #
554 #
555 #
556 #
557 #
558 #
559 #
560 #
561 #
562 #
563 #
564 #
565 #
566 #
567 #
568 #
569 #
570 #
571 #
572 #
573 #
574 #
575 #
576 #
577 #
578 #
579 #
580 #
581 #
582 #
583 #
584 #
585 #
586 #
587 #
588 #
589 #
590 #
591 #
592 #
593 #
594 #
595 #
596 #
597 #
598 #
599 #
600 #
601 #
602 #
603 #
604 #
605 #
606 #
607 #
608 #
609 #
610 #
611 #
612 #
613 #
614 #
615 #
616 #
617 #
618 #
619 #
620 #
621 #
622 #
623 #
624 #
625 #
626 #
627 #
628 #
629 #
630 #
631 #
632 #
633 #
634 #
635 #
636 #
637 #
638 #
639 #
640 #
641 #
642 #
643 #
644 #
645 #
646 #
647 #
648 #
649 #
650 #
651 #
652 #
653 #
654 #
655 #
656 #
657 #
658 #
659 #
660 #
661 #
662 #
663 #
664 #
665 #
666 #
667 #
668 #
669 #
670 #
671 #
672 #
673 #
674 #
675 #
676 #
677 #
678 #
679 #
680 #
681 #
682 #
683 #
684 #
685 #
686 #
687 #
688 #
689 #
690 #
691 #
692 #
693 #
694 #
695 #
696 #
697 #
698 #
699 #
700 #
701 #
702 #
703 #
704 #
705 #
706 #
707 #
708 #
709 #
710 #
711 #
712 #
713 #
714 #
715 #
716 #
717 #
718 #
719 #
720 #
721 #
722 #
723 #
724 #
725 #
726 #
727 #
728 #
729 #
730 #
731 #
732 #
733 #
734 #
735 #
736 #
737 #
738 #
739 #
740 #
741 #
742 #
743 #
744 #
745 #
746 #
747 #
748 #
749 #
750 #
751 #
752 #
753 #
754 #
755 #
756 #
757 #
758 #
759 #
760 #
761 #
762 #
763 #
764 #
765 #
766 #
767 #
768 #
769 #
770 #
771 #
772 #
773 #
774 #
775 #
776 #
777 #
778 #
779 #
780 #
781 #
782 #
783 #
784 #
785 #
786 #
787 #
788 #
789 #
790 #
791 #
792 #
793 #
794 #
795 #
796 #
797 #
798 #
799 #
800 #
801 #
802 #
803 #
804 #
805 #
806 #
807 #
808 #
809 #
810 #
811 #
812 #
813 #
814 #
815 #
816 #
817 #
818 #
819 #
820 #
821 #
822 #
823 #
824 #
825 #
826 #
827 #
828 #
829 #
830 #
831 #
832 #
833 #
834 #
835 #
836 #
837 #
838 #
839 #
840 #
841 #
842 #
843 #
844 #
845 #
846 #
847 #
848 #
849 #
850 #
851 #
852 #
853 #
854 #
855 #
856 #
857 #
858 #
859 #
860 #
861 #
862 #
863 #
864 #
865 #
866 #
867 #
868 #
869 #
870 #
871 #
872 #
873 #
874 #
875 #
876 #
877 #
878 #
879 #
880 #
881 #
882 #
883 #
884 #
885 #
886 #
887 #
888 #
889 #
890 #
891 #
892 #
893 #
894 #
895 #
896 #
897 #
898 #
899 #
900 #
901 #
902 #
903 #
904 #
905 #
906 #
907 #
908 #
909 #
910 #
911 #
912 #
913 #
914 #
915 #
916 #
917 #
918 #
919 #
920 #
921 #
922 #
923 #
924 #
925 #
926 #
927 #
928 #
929 #
930 #
931 #
932 #
933 #
934 #
935 #
936 #
937 #
938 #
939 #
940 #
941 #
942 #
943 #
944 #
945 #
946 #
947 #
948 #
949 #
950 #
951 #
952 #
953 #
954 #
955 #
956 #
957 #
958 #
959 #
960 #
961 #
962 #
963 #
964 #
965 #
966 #
967 #
968 #
969 #
970 #
971 #
972 #
973 #
974 #
975 #
976 #
977 #
978 #
979 #
980 #
981 #
982 #
983 #
984 #
985 #
986 #
987 #
988 #
989 #
990 #
991 #
992 #
993 #
994 #
995 #
996 #
997 #
998 #
999 #
1000 #
```

```

47 # random bits generator
48 def bits_gen(values): # function takes a list of values between 0 and
49     data = []
50
51     for i in values:
52         #if the values is less than 0.5 append a 0
53         if i < 0.5:
54             data.append(0)
55         else:
56             #if the value found in the list is greater than 0.5 append
57             1
58             data.append(1)
59
60     return data
61 #
62 # -----
63 #           mapping of bits to symbol using constellation maps
64 # -----
65
64 def BPSK(bits):
65     bpsk = []
66     for k in bits:
67         if k == 1:
68             bpsk.append(1)
69         else:
70             bpsk.append(-1)
71     return bpsk
72
73 def fourQAM(bits):
74     FQAM = []
75     M = 2
76     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
77     for k in subList:
78         if k == [0, 0]:
79             FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
80         elif k == [0, 1]:
81             FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
82         elif k == [1, 1]:
83             FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
84         # elif(k==[1,0]):
85         elif k == [1, 0]:
86             FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))
87
88     return FQAM
89
90 def eight_PSK(bits):
91     EPSK = []
92     M = 3
93     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
94     for k in subList:
95         if k == [0, 0, 0]:
96             EPSK.append(complex(1, 0))
97         elif k == [0, 0, 1]:

```

```

98         EPSK.append((1+1j)/np.sqrt(2))
99     elif k == [0, 1, 1]:
100         EPSK.append(1j)
101     elif k == [0, 1, 0]:
102         EPSK.append((-1+1j)/np.sqrt(2))
103     elif k == [1, 1, 0]:
104         EPSK.append(-1)
105     elif k == [1, 1, 1]:
106         EPSK.append((-1-1j)/np.sqrt(2))
107     elif k == [1, 0, 1]:
108         EPSK.append(-1j)
109     elif k == [1, 0, 0]:
110         EPSK.append((1-1j)/np.sqrt(2))
111     return EPSK
112
113 # -----
114 #                               Sigma calculation
115 # -----
116
117 def sigma(domain,M):# M is the number of symbols
118     sigma=[]
119     for i in domain:
120         sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M
121 )))
122     return sigma
123
124 # -----
125 #                               create noise
126 # -----
127
128 def noise(size,sigma):
129     noiseList = np.random.normal(0,sigma,size)
130     return noiseList
131
132 # -----
133 #                               add noise
134 # -----
135
136 def addnoise(transmitted,channels,L,m,snr):# assuming transmitted comes
137     with the memory symbols padded
138     recieved=[]
139     M=m
140     k=snr
141
142     sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
143     #print(sigma_)
144

```

```

145     for i in range(L-1,len(transmitted)):
146         sample=np.random.normal(0,1,1)[0]
147         recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*
channels[0+1]
148                             +transmitted[i-2]*channels[2]+sigma_*(sample+(
sample)*1j))
149     return recieved #without the padding
150
151 #
-----
152 #                                     DFE function
153 #
-----
154
155 def DFE(recieved,channels,L,options,memory):
156     Options =options# [1,-1]# these are the option available for bpsk
157     symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
158     s=0 # symbol mover
159     for i in range(0,len(recieved)):
160         guess=[]
161         n=len(channels)-1# length of the chanel L-1
162         #calculating the product but from second position
163         sumof=0
164         for j in range(1,n):
165             sumof+= symbols[n-1+s]*channels[j]
166             n-=1
167
168         for k in Options:
169             guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))
**2)
170         estimate=Options[guess.index(min(guess))]
171         symbols.append(estimate)
172         s+=1
173     return symbols[L-1:] #final
174 #
-----
175 #                                     Options and memory generator
176 #
-----
177 def OptMemGen(i,L):
178     #BPSK=1 #4QAM=2 8PSK=3
179     if i==1:
180         Options = [1,-1]# these are the option available for bpsk
181         memory= [1]*(L-1) #the first 1 is the memory symbols
182         return Options,memory
183     elif(i==2):
184         Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.
sqrt(2), (1-1j)/np.sqrt(2)]
185         memory=[(1+1j)/np.sqrt(2)]*(L-1)
186         return Options,memory
187     elif(i==3):
188         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1,
(-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]

```

```

189     memory=[(1+1j)/np.sqrt(2)]*(L-1)
190     return Options,memory
191
192 #
193 # -----
194 #
195 # Bit Error calculation
196 # -----
197
198 def bit_errors(sent, recieved):
199     error = 0
200     for k in range(0,len(recieved)):
201         if sent[k] != recieved[k]:
202             error += 1
203     BER = error / len(recieved)*100
204     return BER
205
206 #transmitted=[1,1,1,-1,1,-1,1]
207 #channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
208 #Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
209 #print(Recieved)
210 #
211 #####
212
213 """
214 Lefa's graph function
215
216
217
218
219
220
221
222
223
224
225
226
227
228 """
229 #
230 #####
231
232
233 sqrt2 = np.sqrt(2)
234 c = [0.89 + 0.92j, 0.42 - 0.37j, 0.19 + 0.12j]
235 N = 200
236
237

```



```

238 def Reverse(x):
239     return x[::-1]
240
241 # generate deltas for BPSK
242 def findDeltaBPSK(recieved, Symbols =[],i=0):
243     Options = [-1,1]
244     delta = []
245     for s in Symbols:
246         for j in Options:
247             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
248 [0]*c[2]))**2)
249     return delta
250
251 # generate deltas for 4QAM
252 def findDelta4QAM(recieved, Symbols =[],i=0):
253     Options = [(1+1j)/sqrt2, (-1+1j)/sqrt2, (-1-1j)/sqrt2, (1-1j)/
254 sqrt2]
255     delta = []
256     for s in Symbols:
257         for j in Options:
258             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
259 [0]*c[2]))**2)
260     return delta
261
262 # generate deltas for 8PSK
263 def findDelta8PSK(recieved, Symbols =[],i=0):
264     Options = [1, (1+1j)/sqrt2, 1j, (-1+1j)/sqrt2, -1, (-1-1j)/sqrt2,
265 -1j, (1-1j)/sqrt2]
266     delta = []
267     for s in Symbols:
268         for j in Options:
269             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
270 [0]*c[2]))**2)
271     return delta
272
273 def BPSK_MLSE(recieved, N):
274     # Generate the bits, then get their symbols from the constellation
275     map
276     Bits = []
277     for i in range(8):
278         Bits.append(format(8+i, 'b'))
279     Symbols = []
280     # print(Bits)
281     for i in range(8):
282         Sym = []
283         for j in range(3):
284             if Bits[i][j+1] == "1":
285                 Sym.append(1)
286             else:
287                 Sym.append(-1)
288         Symbols.append(Sym)
289     # print(Symbols)
290     # Get all the deltas
291     deltas = []
292     for i in range(N):

```

```

289         deltas.append(findDeltaBPSK(recieved, Symbols ,i))
290
291     # Using the deltas, work backwards and determine the transmitted
sequence
292     transmitted = []
293     for i in range(N):
294         cost = min(deltas[N-1-i])
295         bit = deltas[N-1-i].index(cost)
296
297         # print(bit)
298         if bit % 2 == 0:
299             transmitted.append(-1)
300         else:
301             transmitted.append(1)
302     transmitted = Reverse(transmitted)
303     return transmitted
304
305 def MLSE_4QAM(recieved, N):
306     # Generate the bits, then get their symbols from the constellation
map
307     Bits = []
308     for i in range(64):
309         Bits.append(format(64+i, 'b'))
310     Symbols = []
311     i = 0
312     while i < 64:
313         Sym = []
314         for j in range(6):
315             if Bits[i][j+1:j+3] == "00":
316                 Sym.append((1+1j)/sqrt2)
317             elif Bits[i][j+1:j+3] == "01":
318                 Sym.append((-1+1j)/sqrt2)
319             elif Bits[i][j+1:j+3] == "11":
320                 Sym.append((-1-1j)/sqrt2)
321             elif Bits[i][j+1:j+3] == "10":
322                 Sym.append((1-1j)/sqrt2)
323         Symbols.append(Sym)
324         i += 2
325
326     # Get all the deltas
327     deltas = []
328     for i in range(N):
329         deltas.append(findDelta4QAM(recieved, Symbols ,i))
330
331     # print(deltas)
332     # Using the deltas, work backwards and determine the transmitted
sequence
333     transmitted = []
334     for i in range(N):
335         cost = min(deltas[N-1-i])
336         bit = deltas[N-1-i].index(cost)
337
338         if bit % 4 == 0: #recieved (1+1j)/sqrt2
339             transmitted.append((1+1j)/sqrt2)
340         elif bit % 4 == 1: #recieved (-1+1j)/sqrt2
341             transmitted.append((-1+1j)/sqrt2)
342         elif bit % 4 == 2: #recieved (-1-1j)/sqrt2

```

```

343         transmitted.append((-1-1j)/sqrt2)
344     elif bit % 4 == 3: #recieved (1-1j)/sqrt2
345         transmitted.append((1-1j)/sqrt2)
346
347     transmitted = Reverse(transmitted)
348
349     return transmitted
350
351 def MLSE_8PSK(recieved = [], N = 4):
352     # Generate the bits, then get their symbols from the constellation
353     # map
354     L=3
355     Bits = []
356
357     for i in range(512):
358         Bits.append(format(512+i, 'b'))
359     Symbols = []
360     i = 0
361     Sym = []
362     while i < 8**L:
363         Sym = []
364         for j in range(9):
365             if Bits[i][j+1:j+4] == "111":
366                 Sym.append(1)
367             elif Bits[i][j+1:j+4] == "110":
368                 Sym.append((1+1j)/sqrt2)
369             elif Bits[i][j+1:j+4] == "010":
370                 Sym.append(1j)
371             elif Bits[i][j+1:j+4] == "011":
372                 Sym.append((-1+1j)/sqrt2)
373             elif Bits[i][j+1:j+4] == "001":
374                 Sym.append(-1)
375             elif Bits[i][j+1:j+4] == "000":
376                 Sym.append((-1-1j)/sqrt2)
377             elif Bits[i][j+1:j+4] == "100":
378                 Sym.append(-1j)
379             elif Bits[i][j+1:j+4] == "101":
380                 Sym.append((1-1j)/sqrt2)
381         Symbols.append(Sym)
382         i += 3
383
384     # Get all the deltas
385     deltas = []
386     for i in range(N):
387         deltas.append(findDelta8PSK(recieved, Symbols ,i))
388
389     # print(deltas)
390     # Using the deltas, work backwards and determine the transmitted
391     # sequence
392     transmitted = []
393     for i in range(N):
394         cost = min(deltas[N-1-i])
395         bit = deltas[N-1-i].index(cost)
396
397         if bit % 8 == 0:
398             transmitted.append(1)

```

```

398         elif bit % 8 == 1:
399             transmitted.append((1+1j)/sqrt2)
400         elif bit % 8 == 2:
401             transmitted.append(1j)
402         elif bit % 8 == 3:
403             transmitted.append((-1+1j)/sqrt2)
404         elif bit % 8 == 4:
405             transmitted.append(-1)
406         elif bit % 8 == 5:
407             transmitted.append((-1-1j)/sqrt2)
408         elif bit % 8 == 6:
409             transmitted.append(-1j)
410         elif bit % 8 == 7:
411             transmitted.append((1-1j)/sqrt2)
412
413     transmitted = Reverse(transmitted)
414     return transmitted
415
416 def Graph():
417     size=N
418     randomValues= theorwhichman(size)
419     bits=bits_gen(randomValues)
420     BPSK_bits=BPSK(bits)
421     FourQAM_bits=fourQAM(bits)
422     EBPSK_bits=eight_PSK(bits)
423     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
424     transmitted=[]
425     xValues = np.linspace(-4, 15, 38)
426     yvalues=[]
427     #bpsk
428
429     L=3
430     M=2
431     a,transmitted=OptMemGen(1,3)
432     transmitted.extend(BPSK_bits)
433     k=-4
434     while (k<15):
435
436         Recieved=addnoise(transmitted,channels,L,M,k)#
437         [1.5,1.2,1,-1.2,-1.5,0.2]
438         #print(Recieved)
439         Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
440         Detected=BPSK_MLSE(Recieved,size)
441         yvalues.append(bit_errors(transmitted[L-1:],Detected))
442         k+=0.5
443     plt.semilogy(xValues,yvalues, label="BPSK")
444     plt.ylabel('BER')
445     plt.xlabel('SNR')
446
447
448     yvalues=[]
449
450     #4Qam
451     L=3
452     M=4
453     a,transmitted=OptMemGen(2,3)

```

```

454     transmitted.extend(FourQAM_bits)
455     k=-4
456     while (k<15):
457
458
459         Recieved=addnoise(transmitted,channels,L,M,k)#
[1.5,1.2,1,-1.2,-1.5,0.2]
460         Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
461         Detected=MLSE_4QAM(Recieved,int(size/2))
462         yvalues.append(bit_errors(transmitted[L-1:],Detected))
463         k+=0.5
464
465
466     plt.semilogy(xValues,yvalues, label="4QAM")
467     plt.ylabel('BER')
468     plt.xlabel('SNR')
469     yvalues=[]
470
471     L=3
472     M=8 #BPSK=2 4Qam=4 8psk=8
473     #8psk
474     a,transmitted=OptMemGen(3,3)
475     transmitted.extend(EBPSK_bits)
476
477     k=-4
478     while (k<15):
479
480
481         Recieved=addnoise(transmitted,channels,L,M,k)#
[1.5,1.2,1,-1.2,-1.5,0.2]
482         Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
483         Detected=MLSE_8PSK(Recieved,int(size/3))
484         yvalues.append(bit_errors(transmitted[L-1:],Detected))
485         k+=0.5
486
487     plt.semilogy(xValues,yvalues, label="8PSK")
488     plt.ylabel('BER')
489     plt.xlabel('SNR')
490     plt.title(" BER vs SNR")
491     plt.legend()
492 # print(BPSK_MLSE(r,N))
493 #
494 #print(MLSE_4QAM(r,N))
495 #
496 Graph()

```

### 8.1.2 DFE: Static CIR

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Fri Oct  2 22:11:23 2020
4
5 @author: user
6 """
7
8 import random
9 import math
10 import statistics as st

```

```

11 import numpy as np
12 import matplotlib.pyplot as plt
13 from scipy.stats import norm
14 import seaborn as sns
15 # settings for seaborn plotting style
16 sns.set(color_codes=True)
17 # settings for seaborn plot sizes
18 sns.set(rc={'figure.figsize':(5,5)})
19 np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
20 #np.random.seed()
21 def theorwhichman(size):
22     rand=[]
23     for i in range(0,size):
24         rand.append(random.uniform(0, 1))
25     return rand
26 size=100
27
28 a=theorwhichman(size)
29
30 mu = st.mean(a)
31 sigma = st.stdev(a)
32
33 x = np.linspace(-1, 1, size)
34 a.sort()
35
36
37
38 """ax = sns.distplot(a,
39                     bins=100,
40                     kde=True,
41                     color='skyblue',
42                     hist_kws={"linewidth": 15,'alpha':1})
43 ax.set(xlabel='Normal Distribution', ylabel='Frequency')
44 #[Text(0,0.5,u'Frequency'), Text(0.5,0,u'Normal Distribution')]
45 """
46 #plt.plot(a, norm(0.5, 1).pdf(a))
47 #plt.ylabel('Probability Density')
48 #plt.xlabel('Randomly Generated Numbers')
49 #plt.show()
50
51 print("Sigma:", sigma)
52 print("Mu:", mu)
53
54
55 #Creating bits to be transmitted
56 #
57 -----
58
59 #                                     Bit generator
60 #
61 -----
62
63 # random bits generator
64 def bits_gen(values): # function takes a list of values between 0 and
    1
65     data = []

```

```

62
63     for i in values:
64         #if the values is less than 0.5 append a 0
65         if i < 0.5:
66             data.append(0)
67         else:
68             #if the value found in the list is greater than 0.5 append
69             1
70             data.append(1)
71
72     return data
73 #
74 # -----
75 #           mapping of bits to symbol using constellation maps
76 # -----
77
78 def BPSK(bits):
79     bpsk = []
80     for k in bits:
81         if k == 1:
82             bpsk.append(1)
83         else:
84             bpsk.append(-1)
85     return bpsk
86
87 def fourQAM(bits):
88     FQAM = []
89     M = 2
90     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
91     for k in subList:
92         if k == [0, 0]:
93             FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
94         elif k == [0, 1]:
95             FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
96         elif k == [1, 1]:
97             FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
98         # elif(k==[1,0]):
99         elif k == [1, 0]:
100             FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))
101
102     return FQAM
103
104 def eight_PSK(bits):
105     EPSK = []
106     M = 3
107     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
108     for k in subList:
109         if k == [0, 0, 0]:
110             EPSK.append(complex(1, 0))
111         elif k == [0, 0, 1]:
112             EPSK.append((1+1j)/np.sqrt(2))
113         elif k == [0, 1, 1]:
114             EPSK.append(1j)
115         elif k == [0, 1, 0]:

```

```

114         EPSK.append((-1+1j)/np.sqrt(2))
115     elif k == [1, 1, 0]:
116         EPSK.append(-1)
117     elif k == [1, 1, 1]:
118         EPSK.append((-1-1j)/np.sqrt(2))
119     elif k == [1, 0, 1]:
120         EPSK.append(-1j)
121     elif k == [1, 0, 0]:
122         EPSK.append((1-1j)/np.sqrt(2))
123     return EPSK
124
125 # -----
126 #                               Sigma calculation
127 # -----
128
129 def sigma(domain,M):# M is the number of symbols
130     sigma=[]
131     for i in domain:
132         sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M
133     )))
134     return sigma
135
136 # -----
137 #                               create noise
138 # -----
139
140 def noise(size,sigma):
141     noiseList = np.random.normal(0,sigma,size)
142     return noiseList
143
144 # -----
145 #                               add noise
146 # -----
147
148 def addnoise(transmitted,channels,L,m,snr):# assuming transmitted comes
149     with the memory symbols padded
150     recieved=[]
151     M=m
152     k=snr
153
154     sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
155     #print(sigma_)
156
157     for i in range(L-1,len(transmitted)):
158         sample=np.random.normal(0,1,1)[0]
159         recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*
channels[0+1])

```



```

160         +transmitted[i-2]*channels[2]+sigma_*(sample+(
    sample)*1j))
161     return recieved #without the padding
162
163 #
    -----
164 #                                     DFE function
165 #
    -----
166
167 def DFE(recieved,channels,L,options,memory):
168     Options =options# [1,-1]# these are the option available for bpsk
169     symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
170     s=0 # symbol mover
171     for i in range(0,len(recieved)):
172         guess=[]
173         n=len(channels)-1# length of the chanel L-1
174         #calculating the product but from second position
175         sumof=0
176         for j in range(1,n):
177             sumof+= symbols[n-1+s]*channels[j]
178             n-=1
179
180         for k in Options:
181             #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols
182             [1]*channels[1]+symbols[0]*channels[2]))**2)
183             guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))
184             **2)
185         estimate=Options[guess.index(min(guess))]
186         symbols.append(estimate)
187         s+=1
188     return symbols[L-1:] #final
189 #
    -----
190 #                                     Options and memory generator
191 #
    -----
192
193 def OptMemGen(i,L):
194     #BPSK=1 #4QAM=2 8PSK=3
195     if i==1:
196         Options = [1,-1]# these are the option available for bpsk
197         memory= [1]*(L-1) #the first 1 is the memory symbols
198         return Options,memory
199     elif(i==2):
200         Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.
201         sqrt(2), (1-1j)/np.sqrt(2)]
202         memory=[(1+1j)/np.sqrt(2)]*(L-1)
203         return Options,memory
204     elif(i==3):
205         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1,
206         (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
207         memory=[(1+1j)/np.sqrt(2)]*(L-1)
208         return Options,memory

```

```

204
205 #
-----
206 # Bit Error calculation
207 #
-----

208
209 def bit_errors(sent, recieved):
210     error = 0
211     for k in range(0, len(recieved)):
212         if sent[k] != recieved[k]:
213             error += 1
214     BER = error / len(recieved)*100
215
216     return BER
217
218 #transmitted=[1,1,1,-1,1,-1,1]
219 #channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
220 #Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
221 #print(Recieved)
222
223 def yvalueCal(transmitted):
224     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
225     L=3
226     M=2 #for bpsk=2
227     #SNR=15
228     yvalues=[]
229     #print(transmitted)
230
231     for k in range (-4,16):
232
233
234         Recieved=addnoise(transmitted,channels,L,M,k)#
235         [1.5,1.2,1,-1.2,-1.5,0.2]
236         #print(Recieved)
237         Options,memory= OptMemGen(1,L)
238         Detected=DFE(Recieved,channels,L,Options,memory)
239         yvalues.append(bit_errors(transmitted[L-1:],Detected))
240     return yvalues
241 #print(yvalueCal(transmitted))
242
243 def grapghs():
244     size=100000
245     randomValues= teorwhichman(size)
246     bits=bits_gen(randomValues)
247     BPSK_bits=BPSK(bits)
248     FourQAM_bits=fourQAM(bits)
249     EBPSK_bits=eight_PSK(bits)
250     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
251
252     transmitted=[]
253     xValues = np.linspace(-4, 15, 38)
254     yvalues=[]
255     #bpsk

```

```

256 L=3
257 M=2
258 a,transmitted=OptMemGen(1,3)
259 transmitted.extend(BPSK_bits)
260 k=-4
261 while (k<15):
262
263
264     Recieved=addnoise(transmitted,channels,L,M,k)#
[1.5,1.2,1,-1.2,-1.5,0.2]
265     #print(Recieved)
266     Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
267     Detected=DFE(Recieved,channels,L,Options,memory)
268     yvalues.append(bit_errors(transmitted[L-1:],Detected))
269     k+=0.5
270 plt.semilogy(xValues,yvalues, label="BPSK")
271 plt.ylabel('BER')
272 plt.xlabel('SNR')
273
274
275 yvalues=[]
276 #4Qam
277 L=3
278 M=4
279 a,transmitted=OptMemGen(2,3)
280 transmitted.extend(FourQAM_bits)
281 k=-4
282 while (k<15):
283
284
285     Recieved=addnoise(transmitted,channels,L,M,k)#
[1.5,1.2,1,-1.2,-1.5,0.2]
286     #print(Recieved)
287     Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
288     Detected=DFE(Recieved,channels,L,Options,memory)
289     yvalues.append(bit_errors(transmitted[L-1:],Detected))
290     k+=0.5
291
292
293 plt.semilogy(xValues,yvalues, label="4QAM")
294 plt.ylabel('BER')
295 plt.xlabel('SNR')
296 yvalues=[]
297
298 L=3
299 M=8 #BPSK=2 4Qam=4 8psk=8
300 #8psk
301 a,transmitted=OptMemGen(3,3)
302 transmitted.extend(EBPSK_bits)
303
304 k=-4
305 while (k<15):
306     Recieved=addnoise(transmitted,channels,L,M,k)#
[1.5,1.2,1,-1.2,-1.5,0.2]
307     #print(Recieved)
308     Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
309     Detected=DFE(Recieved,channels,L,Options,memory)

```

```

310         yvalues.append(bit_errors(transmitted[L-1:],Detected))
311         k+=0.5
312
313     plt.semilogy(xValues,yvalues, label="8PSK")
314     plt.ylabel('BER')
315     plt.xlabel('SNR')
316     plt.title(" BER vs SNR")
317     plt.legend()
318
319 graphs() #uncomment this if you want the graphs
320
321
322 def tester():
323     size=10000
324     #np.random.seed(420)
325     print("Generating the random number generator of size 200")
326     randomValues= theorwhichman(size)
327
328
329     print("\nExpected sigma =0.29 and expected mu=0.50")
330     print("Sigma",st.mean(randomValues))
331     print("mu",st.stdev(randomValues))
332
333     print("\nTesting the bits_gen function of size 200")
334     bits=bits_gen(randomValues)
335     print("Size:",len(bits))
336
337     print("\nNow testing the mapping of sysmbols for different
modulation schemes")
338     print("BPSK expected length 200")
339     BPSK_bits=BPSK(bits)
340     print("BPSK length:",len(BPSK_bits))
341     print("4QAM expected length =100")
342     #FourQAM_bits=fourQAM(bits)
343     #print("4QAM length:",len(FourQAM_bits))
344     print("8BPSK expected length 66")
345     #EBPSK_bits=eight_PSK(bits)
346     #print("8BPSK length:",len(EBPSK_bits))
347     #SNR = np.linspace(0, 15, 16)
348     #print(SNR)
349     #print("\nGenerating noise for 8psk")
350     #noiseList=noise(len(EBPSK_bits),sigma(SNR,8)[0])
351     #print(noiseList)
352     #print("Length of noise:",len(noiseList))
353     #print("variance:" ,st.variance(noiseList))
354
355     print("\nTesting the DFE function")
356     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
357     #bpsk
358     transmitted=[1,1]
359     transmitted.extend(BPSK_bits)
360     #QPSK
361
362     L=3
363     M=2 #for bpsk=2
364     SNR=-1
365

```

```

366     print("Adding Noise")
367     #Recieved=addnoise(transmitted,channels,L,M,SNR)
368     #[1.5,1.2,1,-1.2,-1.5,0.2]
369     #Options,memory= OptMemGen(1,L)
370     #print("Received symbols:",Recieved)
371     #Detected=DFE(Recieved,channels,L,Options,memory)
372     #print("Detected Symbols",Detected)
373     xValues = np.linspace(-4, 15, 20)
374     yvalues=yvalueCal(transmitted)
375     print(yvalues)
376     plt.semilogy(xValues,yvalues)
377
378
379 #tester()

```

### 8.1.3 MLSE: Dynamic CIR

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Oct  1 08:28:37 2020
4
5  @author: Kwaku
6  """
7  import random
8  import math
9  import statistics as st
10 import numpy as np
11 import matplotlib.pyplot as plt
12 from scipy.stats import norm
13 import seaborn as sns
14
15
16
17
18 # settings for seaborn plotting style
19 sns.set(color_codes=True)
20 # settings for seaborn plot sizes
21 sns.set(rc={'figure.figsize':(5,5)})
22 np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
23 #np.random.seed()
24 def theorwhichman(size):
25     rand=[]
26     for i in range(0,size):
27         rand.append(random.uniform(0, 1))
28     return rand
29 size=100
30
31 a=theorwhichman(size)
32
33 mu = st.mean(a)
34 sigma = st.stdev(a)
35
36 x = np.linspace(-1, 1, size)
37 a.sort()
38
39 print("Sigma:", sigma)

```

```

40 print("Mu:", mu)
41
42
43 #Creating bits to be transmitted
44 #
-----
45 #                               Bit generator
46 #
-----
47 # random bits generator
48 def bits_gen(values): # function takes a list of values between 0 and
49     1
50     data = []
51     for i in values:
52         #if the values is less than 0.5 append a 0
53         if i < 0.5:
54             data.append(0)
55         else:
56             #if the value found in the list is greater than 0.5 append
57             1
58             data.append(1)
59     return data
60
61 #
-----
62 #                               mapping of bits to symbol using constellation maps
63 #
-----
64 def BPSK(bits):
65     bpsk = []
66     for k in bits:
67         if k == 1:
68             bpsk.append(1)
69         else:
70             bpsk.append(-1)
71     bpsk.append(1)
72     return bpsk
73
74 def fourQAM(bits):
75     FQAM = []
76     M = 2
77     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
78     for k in subList:
79         if k == [0, 0]:
80             FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
81         elif k == [0, 1]:
82             FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
83         elif k == [1, 1]:
84             FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
85         # elif(k==[1,0]):
86         elif k == [1, 0]:

```

```

87         FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))
88     FQAM.append((1+1j)/sqrt2)
89     return FQAM
90
91 def eight_PSK(bits):
92     EPSK = []
93     M = 3
94     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
95     for k in subList:
96         if k == [0, 0, 0]:
97             EPSK.append(complex(1, 0))
98         elif k == [0, 0, 1]:
99             EPSK.append((1+1j)/np.sqrt(2))
100        elif k == [0, 1, 1]:
101            EPSK.append(1j)
102        elif k == [0, 1, 0]:
103            EPSK.append((-1+1j)/np.sqrt(2))
104        elif k == [1, 1, 0]:
105            EPSK.append(-1)
106        elif k == [1, 1, 1]:
107            EPSK.append((-1-1j)/np.sqrt(2))
108        elif k == [1, 0, 1]:
109            EPSK.append(-1j)
110        elif k == [1, 0, 0]:
111            EPSK.append((1-1j)/np.sqrt(2))
112    EPSK.append(1)
113    return EPSK
114
115 # -----
116 #                               Sigma calculation
117 # -----
118
119 def sigma(domain,M):# M is the number of symbols
120     sigma=[]
121     for i in domain:
122         sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M
123     )))
124     return sigma
125
126 # -----
127 #                               create noise
128 # -----
129
130 def noise(size,sigma):
131     noiseList = np.random.normal(0,sigma,size)
132     return noiseList
133
134 # -----
135 #                               add noise
136 # -----

```





```

180 #BPSK=1 #4QAM=2 8PSK=3
181 if i==1:
182     Options = [1,-1]# these are the option available for bpsk
183     memory= [1]*(L-1) #the first 1 is the memory symbols
184     return Options,memory
185 elif(i==2):
186     Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.
187 sqrt(2), (1-1j)/np.sqrt(2)]
188     memory=[(1+1j)/np.sqrt(2)]*(L-1)
189     return Options,memory
190 elif(i==3):
191     Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1,
192 (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
193     memory=[(1+1j)/np.sqrt(2)]*(L-1)
194     return Options,memory
195 #
196 -----
197 # Bit Error calculation
198 #
199 -----
200
201 def bit_errors(sent, recieved):
202     error = 0
203     for k in range(0,len(recieved)):
204         if sent[k] != recieved[k]:
205             error += 1
206     BER = error / len(recieved)*100
207     return BER
208
209 #transmitted=[1,1,1,-1,1,-1,1]
210 #channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
211 #Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
212 #print(Recieved)
213 #
214 #####
215 """
216 Lefa's graph function
217
218
219
220
221
222
223
224
225
226
227
228

```

```

229
230 """
231 #
232
233
234
235 sqrt2 = np.sqrt(2)
236 CIR = [0.89 + 0.92j, 0.42 - 0.37j, 0.19 + 0.12j]
237 N = 200
238
239
240 def Reverse(x):
241     return x[::-1]
242
243 # generate deltas for BPSK
244 def findDeltaBPSK(recieved, Symbols = [], i=0, c = CIR):
245     Options = [-1,1]
246     delta = []
247     for s in Symbols:
248         for j in Options:
249             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
[0]*c[2]))**2)
250     return delta
251
252 # generate deltas for 4QAM
253 def findDelta4QAM(recieved, Symbols = [], i=0, c = CIR):
254     Options = [(1+1j)/sqrt2, (-1+1j)/sqrt2, (-1-1j)/sqrt2, (1-1j)/
sqrt2]
255     delta = []
256     for s in Symbols:
257         for j in Options:
258             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
[0]*c[2]))**2)
259
260     return delta
261
262 # generate deltas for 8PSK
263 def findDelta8PSK(recieved, Symbols = [], i=0, c = CIR):
264     Options = [1, (1+1j)/sqrt2, 1j, (-1+1j)/sqrt2, -1, (-1-1j)/sqrt2,
-1j, (1-1j)/sqrt2]
265     delta = []
266     for s in Symbols:
267         for j in Options:
268             delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
[0]*c[2]))**2)
269     return delta
270
271
272 def BPSK_MLSE(recieved, N, c):
273     # Generate the bits, then get their symbols from the constellation
map
274     Bits = []
275     for i in range(8):
276         Bits.append(format(8+i, 'b'))
277     Symbols = []

```

```

278 # print(Bits)
279 for i in range(8):
280     Sym = []
281     for j in range(3):
282         if Bits[i][j+1] == "1":
283             Sym.append(1)
284         else:
285             Sym.append(-1)
286     Symbols.append(Sym)
287 # print(Symbols)
288 # Get all the deltas
289 deltas = []
290 for i in range(N):
291     deltas.append(findDeltaBPSK(recieved, Symbols ,i, c))
292
293 # Using the deltas, work backwards and determine the transmitted
sequence
294 transmitted = []
295 for i in range(N):
296     cost = min(deltas[N-1-i])
297     bit = deltas[N-1-i].index(cost)
298
299     # print(bit)
300     if bit % 2 == 0:
301         transmitted.append(-1)
302     else:
303         transmitted.append(1)
304 transmitted = Reverse(transmitted)
305 return transmitted
306
307 def MLSE_4QAM(recieved, N, c):
308     # Generate the bits, then get their symbols from the constellation
map
309     Bits = []
310     for i in range(64):
311         Bits.append(format(64+i, 'b'))
312     Symbols = []
313     i = 0
314     while i < 64:
315         Sym = []
316         for j in range(6):
317             if Bits[i][j+1:j+3] == "00":
318                 Sym.append((1+1j)/sqrt2)
319             elif Bits[i][j+1:j+3] == "01":
320                 Sym.append((-1+1j)/sqrt2)
321             elif Bits[i][j+1:j+3] == "11":
322                 Sym.append((-1-1j)/sqrt2)
323             elif Bits[i][j+1:j+3] == "10":
324                 Sym.append((1-1j)/sqrt2)
325         Symbols.append(Sym)
326         i += 2
327
328 # Get all the deltas
329 deltas = []
330 for i in range(N):
331     deltas.append(findDelta4QAM(recieved, Symbols ,i, c))
332

```

```

333     # print(deltas)
334     # Using the deltas, work backwards and determine the transmitted
sequence
335     transmitted = []
336     for i in range(N):
337         cost = min(deltas[N-1-i])
338         bit = deltas[N-1-i].index(cost)
339
340         if bit % 4 == 0: #recieved (1+1j)/sqrt2
341             transmitted.append((1+1j)/sqrt2)
342         elif bit % 4 == 1: #recieved (-1+1j)/sqrt2
343             transmitted.append((-1+1j)/sqrt2)
344         elif bit % 4 == 2: #recieved (-1-1j)/sqrt2
345             transmitted.append((-1-1j)/sqrt2)
346         elif bit % 4 == 3: #recieved (1-1j)/sqrt2
347             transmitted.append((1-1j)/sqrt2)
348
349     transmitted = Reverse(transmitted)
350
351     return transmitted
352
353 def MLSE_8PSK(recieved, N, c):
354     # Generate the bits, then get their symbols from the constellation
map
355     L=3
356     Bits = []
357
358     for i in range(512):
359         Bits.append(format(512+i, 'b'))
360     Symbols = []
361     i = 0
362     Sym = []
363     while i < 8**L:
364         Sym = []
365         for j in range(9):
366             if Bits[i][j+1:j+4] == "111":
367                 Sym.append(1)
368             elif Bits[i][j+1:j+4] == "110":
369                 Sym.append((1+1j)/sqrt2)
370             elif Bits[i][j+1:j+4] == "010":
371                 Sym.append(1j)
372             elif Bits[i][j+1:j+4] == "011":
373                 Sym.append((-1+1j)/sqrt2)
374             elif Bits[i][j+1:j+4] == "001":
375                 Sym.append(-1)
376             elif Bits[i][j+1:j+4] == "000":
377                 Sym.append((-1-1j)/sqrt2)
378             elif Bits[i][j+1:j+4] == "100":
379                 Sym.append(-1j)
380             elif Bits[i][j+1:j+4] == "101":
381                 Sym.append((1-1j)/sqrt2)
382         Symbols.append(Sym)
383         i += 3
384
385
386     # Get all the deltas
387     deltas = []

```

```

388     for i in range(N):
389         deltas.append(findDelta8PSK(recieved, Symbols ,i, c))
390
391     # print(deltas)
392     # Using the deltas, work backwards and determine the transmitted
sequence
393     transmitted = []
394     for i in range(N):
395         cost = min(deltas[N-1-i])
396         bit = deltas[N-1-i].index(cost)
397
398         if bit % 8 == 0:
399             transmitted.append(1)
400         elif bit % 8 == 1:
401             transmitted.append((1+1j)/sqrt2)
402         elif bit % 8 == 2:
403             transmitted.append(1j)
404         elif bit % 8 == 3:
405             transmitted.append((-1+1j)/sqrt2)
406         elif bit % 8 == 4:
407             transmitted.append(-1)
408         elif bit % 8 == 5:
409             transmitted.append((-1-1j)/sqrt2)
410         elif bit % 8 == 6:
411             transmitted.append(-1j)
412         elif bit % 8 == 7:
413             transmitted.append((1-1j)/sqrt2)
414
415     transmitted = Reverse(transmitted)
416     return transmitted
417
418 def Graph():
419     size=N
420     size2=5
421     randomValues= theorwhichman(size)
422     bits=bits_gen(randomValues)
423     BPSK_bits=BPSK(bits)
424     FourQAM_bits=fourQAM(bits)
425     EBPSK_bits=eight_PSK(bits)
426     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
427     transmitted=[]
428     xValues = np.linspace(-4, 15, 38*size2)
429     yvalues=[]
430     """#bpsk
431
432     L=3
433     M=2
434     a,transmitted=OptMemGen(1,3)
435     transmitted.extend(BPSK_bits)
436     k=-4
437     while (k<15):
438
439         for i in range(size2):
440             sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
441             c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6),
442                 (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np

```

```

443         .sqrt(6),
            (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
444         .sqrt(6)]
            Recieved=addnoise(transmitted,channels,L,M,k,sigma_)
445         # [1.5,1.2,1,-1.2,-1.5,0.2]
            Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
446         Detected=BPSK_MLSE(Recieved,size,c)
447         yvalues.append(bit_errors(transmitted[L-1:],Detected))
448         k+= 0.5
449         plt.semilogy(xValues,yvalues, label="BPSK")
450         plt.ylabel('BER')
451         plt.xlabel('SNR')
452
453
454         yvalues=[]
455
456         #4Qam
457         L=3
458         M=4
459         a,transmitted=OptMemGen(2,3)
460         transmitted.extend(FourQAM_bits)
461         k=-4
462         while (k<15):
463             for i in range(size2):
464                 sigma_=1/np.sqrt(math.pow(10, (k/ 10))) * 2 * math.log2(M))
465                 c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
466                 .sqrt(6),
                    (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
467                 .sqrt(6),
                    (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
468                 .sqrt(6)]
                    Recieved=addnoise(transmitted,channels,L,M,k,sigma_)
469                 # [1.5,1.2,1,-1.2,-1.5,0.2]
                    Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
470                 Detected=MLSE_4QAM(Recieved,int(size/2),c)
471                 yvalues.append(bit_errors(transmitted[L-1:],Detected))
472                 k+=0.5
473
474
475         plt.semilogy(xValues,yvalues, label="4QAM")
476         plt.ylabel('BER')
477         plt.xlabel('SNR')
478         yvalues=[]
479         """
480         L=3
481         M=8 #BPSK=2 4Qam=4 8psk=8
482         #8psk
483         a,transmitted=OptMemGen(3,3)
484         transmitted.extend(EBPSK_bits)
485
486         k=-4
487         while (k<15):
488             for i in range(size2):
489                 sigma_=1/np.sqrt(math.pow(10, (k/ 10))) * 2 * math.log2(M))
490                 c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
491                 .sqrt(6),
                    (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np

```

```

    .sqrt(6),
492         (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
    .sqrt(6)]
493     Recieved=addnoise(transmitted,channels,L,M,k, sigma_)#
    [1.5,1.2,1,-1.2,-1.5,0.2]
494     Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
495     Detected=MLSE_8PSK(Recieved,int(size/3),c)
496     yvalues.append(bit_errors(transmitted[L-1:],Detected))
497     k+=0.5
498
499     plt.semilogy(xValues,yvalues, label="8PSK")
500     plt.ylabel('BER')
501     plt.xlabel('SNR')
502     plt.title(" BER vs SNR")
503     plt.legend()
504 # print(BPSK_MLSE(r,N))
505 #
506 #print(MLSE_4QAM(r,N))
507 #
508 Graph()

```

### 8.1.4 DFE Dynamic CIR

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Oct 10 10:52:03 2020
4
5 @author: user
6 """
7
8
9 # -*- coding: utf-8 -*-
10 """
11 Created on Fri Oct 2 22:11:23 2020
12
13 @author: user
14 """
15
16 import random
17 import math
18 import statistics as st
19 import numpy as np
20 import matplotlib.pyplot as plt
21 from scipy.stats import norm
22 import seaborn as sns
23 # settings for seaborn plotting style
24 sns.set(color_codes=True)
25 # settings for seaborn plot sizes
26 sns.set(rc={'figure.figsize':(5,5)})
27 np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
28 #np.random.seed()
29 def theorwhichman(size):
30     rand=[]
31     for i in range(0,size):
32         rand.append(random.uniform(0, 1))
33     return rand
34 size=100

```

```

35
36 a=theorwhichman(size)
37
38 mu = st.mean(a)
39 sigma = st.stdev(a)
40
41 x = np.linspace(-1, 1, size)
42 a.sort()
43
44
45
46 """ax = sns.distplot(a,
47                     bins=100,
48                     kde=True,
49                     color='skyblue',
50                     hist_kws={"linewidth": 15,'alpha':1})
51 ax.set(xlabel='Normal Distribution', ylabel='Frequency')
52 #[Text(0,0.5,u'Frequency'), Text(0.5,0,u'Normal Distribution')]
53 """
54 #plt.plot(a, norm(0.5, 1).pdf(a))
55 #plt.ylabel('Probability Density')
56 #plt.xlabel('Randomly Generated Numbers')
57 #plt.show()
58
59 print("Sigma:", sigma)
60 print("Mu:", mu)
61
62
63 #Creating bits to be transmitted
64 #
65 -----
66
67 #                                     Bit generator
68 #
69 -----
70
71 # random bits generator
72 def bits_gen(values): # function takes a list of values between 0 and
73     1
74     data = []
75
76     for i in values:
77         #if the values is less than 0.5 append a 0
78         if i < 0.5:
79             data.append(0)
80         else:
81             #if the value found in the list is greater than 0.5 append
82             1
83             data.append(1)
84
85     return data
86
87 #
88 -----
89
90 #                                     mapping of bits to symbol using constellation maps
91 #

```



```

84 def BPSK(bits):
85     bpsk = []
86     for k in bits:
87         if k == 1:
88             bpsk.append(1)
89         else:
90             bpsk.append(-1)
91     return bpsk
92
93 def fourQAM(bits):
94     FQAM = []
95     M = 2
96     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
97     for k in subList:
98         if k == [0, 0]:
99             FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
100         elif k == [0, 1]:
101             FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
102         elif k == [1, 1]:
103             FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
104         # elif(k==[1,0]):
105         elif k == [1, 0]:
106             FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))
107
108     return FQAM
109
110 def eight_PSK(bits):
111     EPSK = []
112     M = 3
113     subList = [bits[n:n + M] for n in range(0, len(bits), M)]
114     for k in subList:
115         if k == [0, 0, 0]:
116             EPSK.append(complex(1, 0))
117         elif k == [0, 0, 1]:
118             EPSK.append((1+1j)/np.sqrt(2))
119         elif k == [0, 1, 1]:
120             EPSK.append(1j)
121         elif k == [0, 1, 0]:
122             EPSK.append((-1+1j)/np.sqrt(2))
123         elif k == [1, 1, 0]:
124             EPSK.append(-1)
125         elif k == [1, 1, 1]:
126             EPSK.append((-1-1j)/np.sqrt(2))
127         elif k == [1, 0, 1]:
128             EPSK.append(-1j)
129         elif k == [1, 0, 0]:
130             EPSK.append((1-1j)/np.sqrt(2))
131     return EPSK
132
133 # -----
134 #                               Sigma calculation
135 # -----
136
137 def sigma(domain,M):# M is the number of symbols
138     sigma=[]

```

```

139     for i in domain:
140         sigma.append(1/np.sqrt(math.pow(10, (i/ 10))) * 2 * math.log2(M
141         )))
142     return sigma
143
144 #
145 # -----
146 #
147 #
148 # -----
149 #
150 # -----
151 #
152 # -----
153 #
154 # -----
155 #
156 def addnoise(transmitted,channels,L,m,snr):# assuming transmitted comes
157     with the memory symbols padded
158     recieved=[]
159     M=m
160     k=snr
161
162     sigma_=1/np.sqrt(math.pow(10, (k/ 10))) * 2 * math.log2(M))
163     #print(sigma_)
164
165     for i in range(L-1,len(transmitted)):
166         sample=np.random.normal(0,1,1)[0]
167         recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*
168         channels[0+1]
169         +transmitted[i-2]*channels[2]+sigma_*(sample+(
170         sample)*1j))
171     return recieved #without the padding
172
173 #
174 # -----
175 #
176 # -----
177 #
178 # -----
179 #
180 # -----
181 #
182 # -----
183 #
184 # -----
185 #
186 # -----
187 #
188 # -----
189 #
190 # -----
191 #
192 # -----
193 #
194 # -----
195 #
196 # -----
197 #
198 # -----
199 #
200 # -----
201 #
202 # -----
203 #
204 # -----
205 #
206 # -----
207 #
208 # -----
209 #
210 # -----
211 #
212 # -----
213 #
214 # -----
215 #
216 # -----
217 #
218 # -----
219 #
220 # -----
221 #
222 # -----
223 #
224 # -----
225 #
226 # -----
227 #
228 # -----
229 #
230 # -----
231 #
232 # -----
233 #
234 # -----
235 #
236 # -----
237 #
238 # -----
239 #
240 # -----
241 #
242 # -----
243 #
244 # -----
245 #
246 # -----
247 #
248 # -----
249 #
250 # -----
251 #
252 # -----
253 #
254 # -----
255 #
256 # -----
257 #
258 # -----
259 #
260 # -----
261 #
262 # -----
263 #
264 # -----
265 #
266 # -----
267 #
268 # -----
269 #
270 # -----
271 #
272 # -----
273 #
274 # -----
275 #
276 # -----
277 #
278 # -----
279 #
280 # -----
281 #
282 # -----
283 #
284 # -----
285 #
286 # -----
287 #
288 # -----
289 #
290 # -----
291 #
292 # -----
293 #
294 # -----
295 #
296 # -----
297 #
298 # -----
299 #
300 # -----
301 #
302 # -----
303 #
304 # -----
305 #
306 # -----
307 #
308 # -----
309 #
310 # -----
311 #
312 # -----
313 #
314 # -----
315 #
316 # -----
317 #
318 # -----
319 #
320 # -----
321 #
322 # -----
323 #
324 # -----
325 #
326 # -----
327 #
328 # -----
329 #
330 # -----
331 #
332 # -----
333 #
334 # -----
335 #
336 # -----
337 #
338 # -----
339 #
340 # -----
341 #
342 # -----
343 #
344 # -----
345 #
346 # -----
347 #
348 # -----
349 #
350 # -----
351 #
352 # -----
353 #
354 # -----
355 #
356 # -----
357 #
358 # -----
359 #
360 # -----
361 #
362 # -----
363 #
364 # -----
365 #
366 # -----
367 #
368 # -----
369 #
370 # -----
371 #
372 # -----
373 #
374 # -----
375 #
376 # -----
377 #
378 # -----
379 #
380 # -----
381 #
382 # -----
383 #
384 # -----
385 #
386 # -----
387 #
388 # -----
389 #
390 # -----
391 #
392 # -----
393 #
394 # -----
395 #
396 # -----
397 #
398 # -----
399 #
400 # -----
401 #
402 # -----
403 #
404 # -----
405 #
406 # -----
407 #
408 # -----
409 #
410 # -----
411 #
412 # -----
413 #
414 # -----
415 #
416 # -----
417 #
418 # -----
419 #
420 # -----
421 #
422 # -----
423 #
424 # -----
425 #
426 # -----
427 #
428 # -----
429 #
430 # -----
431 #
432 # -----
433 #
434 # -----
435 #
436 # -----
437 #
438 # -----
439 #
440 # -----
441 #
442 # -----
443 #
444 # -----
445 #
446 # -----
447 #
448 # -----
449 #
450 # -----
451 #
452 # -----
453 #
454 # -----
455 #
456 # -----
457 #
458 # -----
459 #
460 # -----
461 #
462 # -----
463 #
464 # -----
465 #
466 # -----
467 #
468 # -----
469 #
470 # -----
471 #
472 # -----
473 #
474 # -----
475 #
476 # -----
477 #
478 # -----
479 #
480 # -----
481 #
482 # -----
483 #
484 # -----
485 #
486 # -----
487 #
488 # -----
489 #
490 # -----
491 #
492 # -----
493 #
494 # -----
495 #
496 # -----
497 #
498 # -----
499 #
500 # -----
501 #
502 # -----
503 #
504 # -----
505 #
506 # -----
507 #
508 # -----
509 #
510 # -----
511 #
512 # -----
513 #
514 # -----
515 #
516 # -----
517 #
518 # -----
519 #
520 # -----
521 #
522 # -----
523 #
524 # -----
525 #
526 # -----
527 #
528 # -----
529 #
530 # -----
531 #
532 # -----
533 #
534 # -----
535 #
536 # -----
537 #
538 # -----
539 #
540 # -----
541 #
542 # -----
543 #
544 # -----
545 #
546 # -----
547 #
548 # -----
549 #
550 # -----
551 #
552 # -----
553 #
554 # -----
555 #
556 # -----
557 #
558 # -----
559 #
560 # -----
561 #
562 # -----
563 #
564 # -----
565 #
566 # -----
567 #
568 # -----
569 #
570 # -----
571 #
572 # -----
573 #
574 # -----
575 #
576 # -----
577 #
578 # -----
579 #
580 # -----
581 #
582 # -----
583 #
584 # -----
585 #
586 # -----
587 #
588 # -----
589 #
590 # -----
591 #
592 # -----
593 #
594 # -----
595 #
596 # -----
597 #
598 # -----
599 #
600 # -----
601 #
602 # -----
603 #
604 # -----
605 #
606 # -----
607 #
608 # -----
609 #
610 # -----
611 #
612 # -----
613 #
614 # -----
615 #
616 # -----
617 #
618 # -----
619 #
620 # -----
621 #
622 # -----
623 #
624 # -----
625 #
626 # -----
627 #
628 # -----
629 #
630 # -----
631 #
632 # -----
633 #
634 # -----
635 #
636 # -----
637 #
638 # -----
639 #
640 # -----
641 #
642 # -----
643 #
644 # -----
645 #
646 # -----
647 #
648 # -----
649 #
650 # -----
651 #
652 # -----
653 #
654 # -----
655 #
656 # -----
657 #
658 # -----
659 #
660 # -----
661 #
662 # -----
663 #
664 # -----
665 #
666 # -----
667 #
668 # -----
669 #
670 # -----
671 #
672 # -----
673 #
674 # -----
675 #
676 # -----
677 #
678 # -----
679 #
680 # -----
681 #
682 # -----
683 #
684 # -----
685 #
686 # -----
687 #
688 # -----
689 #
690 # -----
691 #
692 # -----
693 #
694 # -----
695 #
696 # -----
697 #
698 # -----
699 #
700 # -----
701 #
702 # -----
703 #
704 # -----
705 #
706 # -----
707 #
708 # -----
709 #
710 # -----
711 #
712 # -----
713 #
714 # -----
715 #
716 # -----
717 #
718 # -----
719 #
720 # -----
721 #
722 # -----
723 #
724 # -----
725 #
726 # -----
727 #
728 # -----
729 #
730 # -----
731 #
732 # -----
733 #
734 # -----
735 #
736 # -----
737 #
738 # -----
739 #
740 # -----
741 #
742 # -----
743 #
744 # -----
745 #
746 # -----
747 #
748 # -----
749 #
750 # -----
751 #
752 # -----
753 #
754 # -----
755 #
756 # -----
757 #
758 # -----
759 #
760 # -----
761 #
762 # -----
763 #
764 # -----
765 #
766 # -----
767 #
768 # -----
769 #
770 # -----
771 #
772 # -----
773 #
774 # -----
775 #
776 # -----
777 #
778 # -----
779 #
780 # -----
781 #
782 # -----
783 #
784 # -----
785 #
786 # -----
787 #
788 # -----
789 #
790 # -----
791 #
792 # -----
793 #
794 # -----
795 #
796 # -----
797 #
798 # -----
799 #
800 # -----
801 #
802 # -----
803 #
804 # -----
805 #
806 # -----
807 #
808 # -----
809 #
810 # -----
811 #
812 # -----
813 #
814 # -----
815 #
816 # -----
817 #
818 # -----
819 #
820 # -----
821 #
822 # -----
823 #
824 # -----
825 #
826 # -----
827 #
828 # -----
829 #
830 # -----
831 #
832 # -----
833 #
834 # -----
835 #
836 # -----
837 #
838 # -----
839 #
840 # -----
841 #
842 # -----
843 #
844 # -----
845 #
846 # -----
847 #
848 # -----
849 #
850 # -----
851 #
852 # -----
853 #
854 # -----
855 #
856 # -----
857 #
858 # -----
859 #
860 # -----
861 #
862 # -----
863 #
864 # -----
865 #
866 # -----
867 #
868 # -----
869 #
870 # -----
871 #
872 # -----
873 #
874 # -----
875 #
876 # -----
877 #
878 # -----
879 #
880 # -----
881 #
882 # -----
883 #
884 # -----
885 #
886 # -----
887 #
888 # -----
889 #
890 # -----
891 #
892 # -----
893 #
894 # -----
895 #
896 # -----
897 #
898 # -----
899 #
900 # -----
901 #
902 # -----
903 #
904 # -----
905 #
906 # -----
907 #
908 # -----
909 #
910 # -----
911 #
912 # -----
913 #
914 # -----
915 #
916 # -----
917 #
918 # -----
919 #
920 # -----
921 #
922 # -----
923 #
924 # -----
925 #
926 # -----
927 #
928 # -----
929 #
930 # -----
931 #
932 # -----
933 #
934 # -----
935 #
936 # -----
937 #
938 # -----
939 #
940 # -----
941 #
942 # -----
943 #
944 # -----
945 #
946 # -----
947 #
948 # -----
949 #
950 # -----
951 #
952 # -----
953 #
954 # -----
955 #
956 # -----
957 #
958 # -----
959 #
960 # -----
961 #
962 # -----
963 #
964 # -----
965 #
966 # -----
967 #
968 # -----
969 #
970 # -----
971 #
972 # -----
973 #
974 # -----
975 #
976 # -----
977 #
978 # -----
979 #
980 # -----
981 #
982 # -----
983 #
984 # -----
985 #
986 # -----
987 #
988 # -----
989 #
990 # -----
991 #
992 # -----
993 #
994 # -----
995 #
996 # -----
997 #
998 # -----
999 #
1000 # -----

```

```

180     for i in range(0,len(recieved)):
181         guess=[]
182         n=len(channels)-1# length of the chanel L-1
183         #calculating the product but from second position
184         sumof=0
185         for j in range(1,n):
186             sumof+= symbols[n-1+s]*channels[j]
187             n-=1
188
189         for k in Options:
190             #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols
191             [1]*channels[1]+symbols[0]*channels[2]))**2)
192             guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof)
193             **2))
194             #print(guess[0])
195             estimate=Options[guess.index(min(guess))]
196             symbols.append(estimate)
197             s+=1
198         return symbols[L-1:] #final
199 #
200 -----
201 #
202 #
203 #
204 #
205 #
206 #
207 #
208 #
209 #
210 #
211 #
212 #
213 #
214 #
215 #
216 #
217 #
218 #
219 #
220 #
221 #
222 #
223 #
224 #

```

---

```

198 #
199 #
200 #
201 #
202 #
203 #
204 #
205 #
206 #
207 #
208 #
209 #
210 #
211 #
212 #
213 #
214 #
215 #
216 #
217 #
218 #
219 #
220 #
221 #
222 #
223 #
224 #

```

---

```

200 def OptMemGen(i,L):
201     #BPSK=1 #4QAM=2 8PSK=3
202     if i==1:
203         Options = [1,-1]# these are the option available for bpsk
204         memory= [1]*(L-1) #the first 1 is the memory symbols
205         return Options,memory
206     elif(i==2):
207         Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.
208 sqrt(2), (1-1j)/np.sqrt(2)]
209         memory=[(1+1j)/np.sqrt(2)]*(L-1)
210         return Options,memory
211     elif(i==3):
212         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1,
213 (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
214         memory=[(1+1j)/np.sqrt(2)]*(L-1)
215         return Options,memory
216 #
217 -----
218 #
219 #
220 #
221 #
222 #
223 #
224 #
225 #
226 #
227 #
228 #
229 #
230 #
231 #
232 #
233 #
234 #
235 #
236 #
237 #
238 #
239 #
240 #
241 #
242 #
243 #
244 #

```

---

```

218 #
219 #
220 #
221 #
222 #
223 #
224 #
225 #
226 #
227 #
228 #
229 #
230 #
231 #
232 #
233 #
234 #
235 #
236 #
237 #
238 #
239 #
240 #
241 #
242 #
243 #
244 #

```

---

```

218 def bit_errors(sent, recieved):
219     error = 0
220     for k in range(0,len(recieved)):
221         if sent[k] != recieved[k]:
222             error += 1
223     BER = error / len(recieved)*100
224

```

```

225
226     return BER
227
228 #transmitted=[1,1,1,-1,1,-1,1]
229 #channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
230 #Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
231 #print(Recieved)
232
233 def yvalueCal(transmitted):
234     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
235     L=3
236     M=2 #for bpsk=2
237     #SNR=15
238     yvalues=[]
239     #print(transmitted)
240
241     for k in range (-4,16):
242
243
244         Recieved=addnoise(transmitted,channels,L,M,k)#
245         [1.5,1.2,1,-1.2,-1.5,0.2]
246         #print(Recieved)
247         Options,memory= OptMemGen(1,L)
248         Detected=DFE(Recieved,channels,L,Options,memory)
249         yvalues.append(bit_errors(transmitted[L-1:],Detected))
250     return yvalues
251 #print(yvalueCal(transmitted))
252
253 def newchannel(v1,v2,v3):
254     c=[]
255     b=[v1+v2*1j,(v2+v3*1j),(v3+v1*1j)]/np.sqrt(2.3)
256     c.extend(b)
257     return c
258
259 def graphs():
260     size=1000000
261     randomValues= theorwhichman(size)
262     bits=bits_gen(randomValues)
263     BPSK_bits=BPSK(bits)
264     FourQAM_bits=fourQAM(bits)
265     EBPSK_bits=eight_PSK(bits)
266     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
267
268     transmitted=[]
269     xValues = np.linspace(-4, 15, 38)
270     yvalues=[]
271     #bpsk
272
273     L=3
274     M=2
275     blocks=[BPSK_bits[n:n + 200] for n in range(0, len(BPSK_bits),
276     200)]
277     a,transmitted=OptMemGen(1,3)
278     #transmitted.extend(BPSK_bits)
279     a,transmitter=OptMemGen(1,3)
280     transmitter.extend(BPSK_bits)

```

```

280     k=-4
281     while (k<15):
282         Recieved=[]
283         Detected=[]
284
285         for block in blocks:
286             v1=np.random.normal(0,1,1)[0]
287             v2= np.random.normal(0,1,1)[0]
288             v3 =np.random.normal(0,1,1)[0]
289             channels= newchannel(v1,v2,v3)
290             a,transmitted=OptMemGen(1,3)
291             transmitted.extend(block)
292
293             Recieved=(addnoise(transmitted,channels,L,M,k))
294             Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
295             Detected.extend(DFE(Recieved,channels,L,Options,memory))
296
297             yvalues.append(bit_errors(transmitter[L-1:],Detected))
298             k+=0.5
299     plt.semilogy(xValues,yvalues, label="BPSK")
300     plt.ylabel('BER')
301     plt.xlabel('SNR')
302
303
304     yvalues=[]
305     #4Qam
306     L=3
307     M=4
308     blocks=[FourQAM_bits[n:n + 200] for n in range(0, len(FourQAM_bits)
309 ), 200)]
309     a,transmitter=OptMemGen(2,3)
310     transmitter.extend(FourQAM_bits)
311
312     k=-4
313     while (k<15):
314         Recieved=[]
315         Detected=[]
316
317         for block in blocks:
318             v1=np.random.normal(0,1,1)[0]
319             v2= np.random.normal(0,1,1)[0]
320             v3 =np.random.normal(0,1,1)[0]
321             channels= newchannel(v1,v2,v3)
322             a,transmitted=OptMemGen(2,3)
323             transmitted.extend(block)
324
325             Recieved=(addnoise(transmitted,channels,L,M,k))
326             Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
327             Detected.extend(DFE(Recieved,channels,L,Options,memory))
328
329             yvalues.append(bit_errors(transmitter[L-1:],Detected))
330             k+=0.5
331     plt.semilogy(xValues,yvalues, label="4QAM")
332     plt.ylabel('BER')
333     plt.xlabel('SNR')
334
335     yvalues=[]

```

```

336
337     L=3
338     M=8 #BPSK=2 4Qam=4 8psk=8
339     #8psk
340     blocks=[EBPSK_bits[n:n + 200] for n in range(0, len(EBPSK_bits),
341 200)]
342     a,transmitter=OptMemGen(3,3)
343     transmitter.extend(EBPSK_bits)
344
345     k=-4
346     while (k<15):
347         Recieved=[]
348         Detected=[]
349
350         for block in blocks:
351             v1=np.random.normal(0,1,1)[0]
352             v2= np.random.normal(0,1,1)[0]
353             v3 =np.random.normal(0,1,1)[0]
354             channels= newchannel(v1,v2,v3)
355             a,transmitted=OptMemGen(3,3)
356             transmitted.extend(block)
357
358             Recieved=(addnoise(transmitted,channels,L,M,k))
359             Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
360             Detected.extend(DFE(Recieved,channels,L,Options,memory))
361
362             yvalues.append(bit_errors(transmitter[L-1:],Detected))
363             k+=0.5
364
365     plt.semilogy(xValues,yvalues, label="8PSK")
366     plt.ylabel('BER')
367     plt.xlabel('SNR')
368     plt.title(" BER vs SNR")
369     plt.legend()
370
371
372
373 graphs()
374
375
376 def tester():
377     size=10000
378     #np.random.seed(420)
379     print("Generating the random number generator of size 200")
380     randomValues= theorwhichman(size)
381
382     print("\nExpected sigma =0.29 and expected mu=0.50")
383     print("Sigma",st.mean(randomValues))
384     print("mu",st.stdev(randomValues))
385
386     print("\nTesting the bits_gen function of size 200")
387     bits=bits_gen(randomValues)
388     print("Size:",len(bits))
389
390     print("\nNow testing the mapping of sysmbols for different
modulation schemes")
391     print("BPSK expected length 200")
392     BPSK_bits=BPSK(bits)

```

```

391     print("BPSK length:",len(BPSK_bits))
392     print("4QAM expected length =100")
393     #FourQAM_bits=fourQAM(bits)
394     #print("4QAM length:",len(FourQAM_bits))
395     print("8BPSK expected length 66")
396     #EBPSK_bits=eight_PSK(bits)
397     #print("8BPSK length:",len(EBPSK_bits))
398     #SNR = np.linspace(0, 15, 16)
399     #print(SNR)
400     #print("\nGenerating noise for 8psk")
401     #noiseList=noise(len(EBPSK_bits),sigma(SNR,8)[0])
402     #print(noiseList)
403     #print("Length of noise:",len(noiseList))
404     #print("variance:" ,st.variance(noiseList))
405
406     print("\nTesting the DFE function")
407     channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
408     #bpsk
409     transmitted=[1,1]
410     transmitted.extend(BPSK_bits)
411     #QPSK
412
413     L=3
414     M=2 #for bpsk=2
415     SNR=-1
416
417     print("Adding Noise")
418     #Recieved=addnoise(transmitted,channels,L,M,SNR)
419     # [1.5,1.2,1,-1.2,-1.5,0.2]
420     #Options,memory= OptMemGen(1,L)
421     #print("Received symbols:",Recieved)
422     #Detected=DFE(Recieved,channels,L,Options,memory)
423     #print("Detected Symbols",Detected)
424     xValues = np.linspace(-4, 15, 20)
425     yvalues=yvalueCal(transmitted)
426     print(yvalues)
427     plt.semilogy(xValues,yvalues)
428
429
430 #tester()

```