Japan Advanced Institute of Science and Technology

Homework 09

AWGN Channel

- I232 Information Theory 2021 -

By PRASETYO, Yusuf

2110148

May 29, 2021

Contents

1 Number 1

AWGN Channel:

$$\mathsf{Y} = \mathsf{X} + \mathsf{Z} \tag{1}$$

where $\mathsf{Z} \sim \mathcal{N}(0, \sigma^2)$. Signal-to-noise ratio is defined as

$$\mathrm{SNR} = -10 \log_{10} \sigma^2 \mathrm{dB} \tag{2}$$

Plot of capacity curves with respect to SNR $\in [-10, 20]$ dB

(a) Gaussian: For $\mathsf{X} \in \mathcal{N}(0, 1)$

(b) BPSK: For $\mathsf{X} \in \{-1, +1\}$ with uniform distribution

(c) 4PAM: For $\mathsf{X} \in \{\frac{-3}{\sqrt{5}}, \frac{-1}{\sqrt{5}}, \frac{+1}{\sqrt{5}}, \frac{+3}{\sqrt{5}}\}$ with uniform distribution

(d) 8PAM: For $\mathsf{X} \in \{\frac{-7}{\sqrt{21}}, \frac{-5}{\sqrt{21}}, \frac{-3}{\sqrt{21}}, \frac{-1}{\sqrt{21}}, \frac{+1}{\sqrt{21}}, \frac{+3}{\sqrt{21}}, \frac{+5}{\sqrt{21}}, \frac{+7}{\sqrt{21}}\}$ with uniform distribution

Answer:

(a) For Gaussian input, we can use Gaussian channel capacity theorem

$$C = 0.5 \log\left(1 + \frac{P}{N}\right) \tag{3}$$

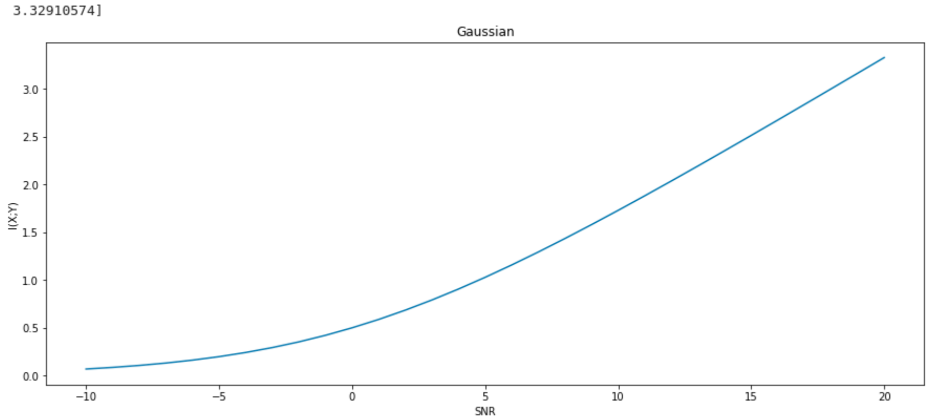
where we calculate $N = \sigma^2$ from the following formula

$$N = 10^{-\mathrm{SNR}/10} \tag{4}$$

and we substitute $P = 1$, the plot becomes

```
k = np.linspace(-10, 20, 31)
v = 10**(-k/10)
I = np.zeros(len(k))
for l in range(len(k)):
    I[l] = 0.5 * np.log(1 + 1/v[l]) / np.log(2)
print(I)
plt.figure(figsize=(15,6))
plt.plot(k,I)
plt.title("Gaussian")
plt.xlabel("SNR")
plt.ylabel("I(X;Y)")
plt.show()
```

```
[0.06875176 0.08553457 0.10612237 0.13123235 0.16164966 0.19820458
 0.24173748 0.29305196 0.35285953 0.42172191 0.5        0.58781832
 0.68505233 0.79134118 0.9061231  1.0286866  1.15822809 1.29390719
 1.43489361 1.58040221 1.72971581 1.88219718 2.03729262 2.19452948
 2.35351013 2.51390384 2.67543808 2.83788995 3.00107822 3.16485623
 3.32910574]
```



(b) For (b)-(d), we will use this following code

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

```
def Gauss(m,v,x):
    """
    Gaussian function
    Needs numpy as np
    Input: mean and variance
    Output: Gaussian that meets the description at position x
    """
    return np.exp(-(x-m)**2/(2*v))/(np.sqrt(2*np.pi*v))
def lGauss(m,v,x):
    """
    Gaussian function
    Needs numpy as np
    Input: mean and variance
    Output: Log( Gaussian that meets the description at position x )
    """
    return (-(x-m)**2/(2*v))-np.log((np.sqrt(2*np.pi*v)))
```

```
def Integrate(f, low, up, N, pxi, xi, px, pygx, lpygx, x, SNR, Gauss, lGauss):
    """
    Integrate a function using trapezoidal rule
    Needs numpy as np
    Input: function to be integrated, lower bound, upper bound, number of increments
    Output: Result of integration (must be a number)
    """
    pts = np.linspace(low, up, N)
    sums = 0
    for pt in pts:
        sums = sums + f(pt, pxi, xi, px, pygx, lpygx, x, SNR, Gauss, lGauss)
    sums = sums - f(low, pxi, xi, px, pygx, lpygx, x, SNR, Gauss, lGauss)/2 - f(up, pxi, xi, px, pygx, lpygx, x, SNR, Gauss, lGauss)/2
    sums = sums*(up-low)/N
    return sums
```

```
def pygx(yi, xi, SNR, Gauss):
    """
    Assumes AWGN out from input x
    Needs numpy as np
    Input: x value
    Output: probabilty of y given x in y
    """
    v = 10**(-SNR/10)
    return Gauss(xi,v,yi)
def lpygx(yi, xi, SNR, lGauss):
    """
    Assumes AWGN out from input x
    Needs numpy as np
    Input: x value
    Output: log( probabilty of y given x in y )
    """
    v = 10**(-SNR/10)
    return lGauss(xi,v,yi)
```

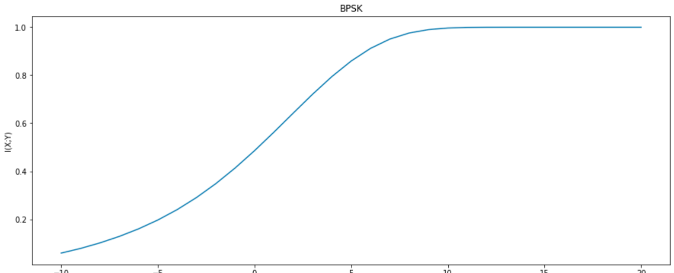
```
def f(yi, pxi, xi, px, pygx, lpygx, x, SNR, Gauss, lGauss):
    """
    Function to be integrated
    Needs numpy as np
    Input: component of function
    Output: output of the function at a certain point
    """
    den = 0
    for j in range(len(px)):
        den = den + px[j]*pygx(yi, x[j], SNR, Gauss)
    lpygxi = lpygx(yi, xi, SNR, lGauss)
    pygxi = pygx(yi, xi, SNR, Gauss)
    return pxi*pygxi*(lpygxi-np.log(den))/np.log(2)
```

```
def MI_disc_cont(x, px, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss):
    """
    Mutual information
    Needs numpy as np
    Input: x discrete, y continuous, z continuous
    Outputs: value of mutual information
    """
    insum = 0
    for i in range(len(x)):
        insum = insum + Integrate(f, low, up, N, px[i], x[i], px, pygx, lpygx, x, SNR, Gauss, lGauss)
    return insum
```

Plugging in for BPSK, we get

```
x = np.array([-1, 1])
px = np.array([0.5, 0.5])
N = 2000
k = np.linspace(-10, 20, 31)
I = np.zeros(len(k))
for l in range(len(k)):
    up = 4.5 + 0.1*(20-k[l])
    low = -4.5 - 0.1*(20-k[l])
    I[l] = MI_disc_cont(x, px, pygx, lpygx, Integrate, low, up, N, k[l], Gauss, lGauss)
print(I)
plt.figure(figsize=(15,6))
plt.plot(k,I)
plt.title("BPSK")
plt.xlabel("SNR")
plt.ylabel("I(X;Y)")
plt.show()
```

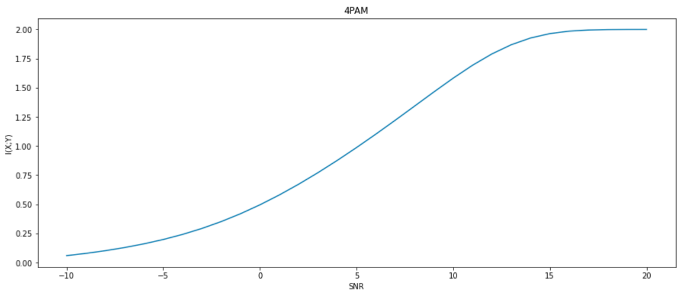
```
[0.05962943 0.07894931 0.10192829 0.12887526 0.16037786 0.19728699
 0.24053115 0.29087097 0.34870144 0.41390406 0.48570116 0.56250674
 0.64182757 0.72030056 0.79395624 0.85876449 0.91142451 0.95020577
 0.97549198 0.98966851 0.99625795 0.99868468 0.99935399 0.99948292
 0.99949883 0.99949996 0.9995     0.9995     0.9995     0.9995
 0.9995    ]
```



(c) Plugging in for 4PAM, we get

```
x = np.array([-3/np.sqrt(5), -1/np.sqrt(5), 1/np.sqrt(5), 3/np.sqrt(5)])
px = np.array([0.25, 0.25, 0.25, 0.25])
N = 2000
k = np.linspace(-10, 20, 31)
I = np.zeros(len(k))
for l in range(len(k)):
    up = 4.5 + 0.1*(20-k[l])
    low = -4.5 - 0.1*(20-k[l])
    I[l] = MI_disc_cont(x, px, pygx, lpygx, Integrate, low, up, N, k[l], Gauss, lGauss)
print(I)
plt.figure(figsize=(15,6))
plt.plot(k,I)
plt.title("4PAM")
plt.xlabel("SNR")
plt.ylabel("I(X;Y)")
plt.show()
```

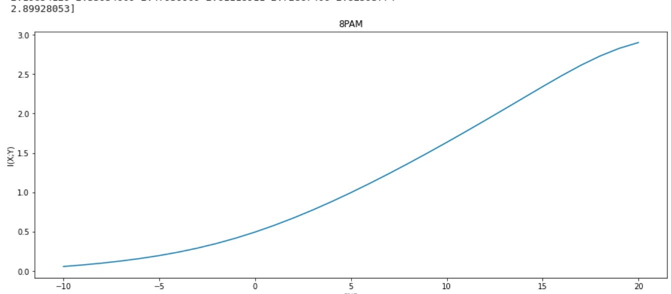
```
[0.05907483 0.0783058  0.10129306 0.12837067 0.16011125 0.19733833
 0.2410064  0.29201925 0.35107644 0.41858484 0.49462313 0.57894456
 0.6710202  0.77012278 0.87542776 0.98609038 1.10126587 1.22004148
 1.34119707 1.46270937 1.58118061 1.691736   1.78880799 1.86759383
 1.92554865 1.96328372 1.984403   1.99419845 1.99779625 1.99878527
 1.998975  ]
```



(d) Plugging in for 8PAM, we get

```
x = np.array([-7/(np.sqrt(21)), -5/(np.sqrt(21)), -3/(np.sqrt(21)), -1/(np.sqrt(21)),
              1/(np.sqrt(21)), 3/(np.sqrt(21)), 5/(np.sqrt(21)), 7/(np.sqrt(21))])
px = np.array([1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8, 1/8])
N = 2000
k = np.linspace(-10, 20, 31)
I = np.zeros(len(k))
up = 5
low = -5
for l in range(len(k)):
    up = 4.5 + 0.1*(20-k[l])
    low = -4.5 - 0.1*(20-k[l])
    I[l] = MI_disc_cont(x, px, pygx, lpygx, Integrate, low, up, N, k[l], Gauss, lGauss)
print(I)
plt.figure(figsize=(15,6))
plt.plot(k,I)
plt.title("8PAM")
plt.xlabel("SNR")
plt.ylabel("I(X;Y)")
plt.show()
```

```
[0.05896981 0.07818114 0.10116459 0.12825961 0.16003782 0.19731786
 0.24105373 0.29216203 0.35137385 0.41914832 0.49563789 0.58069037
 0.67388677 0.77461506 0.8821641  0.99580866 1.1148627  1.23869916
 1.36674978 1.49849781 1.63346904 1.77122124 1.91132793 2.05332895
 2.19654129 2.33954609 2.47939909 2.61118911 2.72867406 2.82593774
 2.89928053]
```



2 Number 2

We first write the probability constraint

$$\sum_{x\in\mathcal{X}} p_{\mathsf{X}}(x) = 2p_a + 2p_b = 1 \tag{5}$$

and the power constraint is

$$\sum_{x\in\mathcal{X}} x^2 p_{\mathsf{X}}(x) = 2a^2 p_a + 2b^2 p_b \leq 1 \tag{6}$$

Mutual information of the system is

$$\sum_{x\in\mathcal{X}} \int_{-\infty}^{\infty} dy\, p_{\mathsf{X}}(x) p_{\mathsf{Y}|\mathsf{X}}(y|x) \log\left(\frac{p_{\mathsf{Y}|\mathsf{X}}(y|x)}{p_{\mathsf{Y}}(y)}\right) \tag{7}$$

(a) We then edit our previous code. We used gradient descent optimization as elaborated below.

```python
import numpy as np
import matplotlib.pyplot as plt
import math
```

```python
def Gauss(m,v,x):
    """
    Gaussian function
    Needs numpy as np
    Input: mean and variance
    Output: Gaussian that meets the description at position x
    """
    return np.exp(-(x-m)**2/(2*v))/(np.sqrt(2*np.pi*v))
def lGauss(m,v,x):
    """
    Gaussian function
    Needs numpy as np
    Input: mean and variance
    Output: Log( Gaussian that meets the description at position x )
    """
    return (-(x-m)**2/(2*v))-np.log((np.sqrt(2*np.pi*v)))
```

```python
def Integrate(f2, low, up, N, pxi, xi, pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa):
    """
    Integrate a function using trapezoidal rule
    Needs numpy as np
    Input: function to be integrated, lower bound, upper bound, number of increments
    Output: Result of integration (must be a number)
    """
    pts = np.linspace(low, up, N)
    sums = 0
    for pt in pts:
        sums = sums + f2(pt, pxi, xi, pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa)
    sums = sums - f2(low, pxi, xi, pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa)/2 - f2(up, pxi, xi, pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa)/2
    sums = sums*(up-low)/N
    return sums
```

```python
def pygx(yi, xi, SNR, Gauss):
    """
    Assumes AWGN out from input x
    Needs numpy as np
    Input: x value
    Output: probabilty of y given x in y
    """
    v = 10**(-SNR/10)
    return Gauss(xi,v,yi)
def lpygx(yi, xi, SNR, lGauss):
    """
    Assumes AWGN out from input x
    Needs numpy as np
    Input: x value
    Output: log( probabilty of y given x in y )
    """
    v = 10**(-SNR/10)
    return lGauss(xi,v,yi)
```

```python
def xs(a, pa):
    return np.array([-np.sqrt((0.5-(a**2)*pa)/(0.5-pa)),-a,a,np.sqrt((0.5-(a**2)*pa)/(0.5-pa))])
```

```python
def pxs(pa):
    return np.array([0.5-pa,pa,pa,0.5-pa])
```

```python
def f2(yi, pxi, xi, pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa):
    """
    Function to be integrated
    Needs numpy as np
    Input: component of function
    Output: output of the function at a certain point
    """
    den = 0
    xss = xs(a, pa)
    pxss = pxs(pa)
    for j in range(len(xss)):
        den = den + pxss[j]*pygx(yi, xss[j], SNR, Gauss)
    lpygxi = lpygx(yi, xi, SNR, lGauss)
    pygxi = pygx(yi, xi, SNR, Gauss)
    return pxi*pygxi*(lpygxi-np.log(den))/np.log(2)
```

```python
def MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a, pa):
    """
    Mutual information
    Needs numpy as np
    Input: x discrete, y continuous, z continuous
    Outputs: value of mutual information
    """
    insum = 0
    xss = xs(a, pa)
    pxss = pxs(pa)
    for i in range(len(xss)):
        insum = insum + Integrate(f2, low, up, N, pxss[i], xss[i], pxs, pygx, lpygx, xs, SNR, Gauss, lGauss, a, pa)
    return insum
```

```
def grad_descent(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a, pa, numdif, step):
    """
    Gradient Descent of MI_disc_cont
    Input: starting a and pa; hyperparameter numdif and step
    Output: stable a and pa which minimizes MI_disc_cont
    """
    da = 0
    dpa = 0
    starting = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a, pa)
    ga = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+numdif, pa)
    gpa = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a, pa+numdif)
    va = (ga - starting)/numdif
    vpa = (gpa - starting)/numdif
    print("a =", a)
    print("pa =", pa)
    print("b =", np.sqrt((0.5-(a**2)*pa)/(0.5-pa)))
    print("pb =", 0.5-pa)
    prev = 0
    while prev < starting:
        hatva = va/(np.sqrt(va**2+vpa**2))
        hatvpa = vpa/(np.sqrt(va**2+vpa**2))
        prev = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+da, pa+dpa)
        da += step*hatva
        dpa += step*hatvpa
        va_prev = (ga - starting)/numdif
        vpa_prev = (gpa - starting)/numdif
        starting = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+da, pa+dpa)
        ga = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+da+numdif, pa+dpa)
        gpa = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+da, pa+dpa+numdif)
        va = (ga - starting)/numdif
        vpa = (gpa - starting)/numdif
        print("va =", va)
        print("vpa =", vpa)
        print("a =", a+da)
        print("pa =", pa+dpa)
        print(starting)
    da -= step*hatva
    dpa -= step*hatvpa
    starting = MI_disc_cont(xs, pxs, pygx, lpygx, Integrate, low, up, N, SNR, Gauss, lGauss, a+da, pa+dpa)
    return np.array([starting, a+da, pa+dpa])
```

```
a = 0.8
pa = 0.25
N = 2000
k = np.linspace(-10, 20, 31)
aas = np.zeros(len(k))
paas = np.zeros(len(k))
Hs = np.zeros(len(k))
for l in range(len(k)):
    up = 4.5 + 0.1*(20-k[l])
    low = -4.5 - 0.1*(20-k[l])
    res = grad_descent(xs, pxs, pygx, lpygx, Integrate, low, up, N, k[l], Gauss, lGauss, a, pa, numdif=0.003, step=0.003)
    print("SNR =", k[l])
    print(res)
    Hs[l] = res[0]
    aas[l] = res[1]
    paas[l] = res[2]
```

```
pa = 0.24933079260056196
1.9985114066847998
va = -0.012960644168410104
vpa = -0.0002394162090801899
a = 0.5571931302548503
pa = 0.24927141401183614
1.9985481457947634
va = -0.011983586818864467
vpa = -0.0002064714564505484
a = 0.5541936419768096
pa = 0.24921600580037861
1.9985821016239196
```

and many lines later

```
va = -0.006439568112970306
vpa = 0.04445750606355917
a = 0.5281831875518098
pa = 0.24696594789024992
1.9986768364470056
SNR = 20.0
[1.99878377 0.53021021 0.24917755]
```

```
paas
```

```
array([0.15625087, 0.15605969, 0.15569619, 0.15497831, 0.15278442,
       0.18520404, 0.31399207, 0.42727334, 0.43254399, 0.43337573,
       0.43280065, 0.43170492, 0.4293682 , 0.4249057 , 0.41411814,
       0.39637226, 0.38469013, 0.37146798, 0.35784704, 0.34304851,
       0.32740606, 0.31234982, 0.29826441, 0.28465189, 0.27292824,
       0.26296169, 0.25590367, 0.25164267, 0.24974955, 0.24939316,
       0.24917755])
```

```
aas
```

```
array([0.99923105, 0.99912615, 0.99892631, 0.99853019, 1.00035605,
       0.05154951, 0.2598613 , 0.67464115, 0.67282981, 0.67362057,
       0.67272133, 0.67105133, 0.66836085, 0.66251037, 0.6358882 ,
       0.59571804, 0.57517716, 0.55402713, 0.53542485, 0.51690734,
       0.49878002, 0.48403956, 0.47272497, 0.46189427, 0.45427592,
       0.44687628, 0.44261637, 0.44156686, 0.44079026, 0.47933118,
       0.53021021])
```

```
Hs
```

```
array([0.05962943, 0.07894931, 0.10192829, 0.12887526, 0.16037786,
       0.19735094, 0.24122624, 0.29268961, 0.35260108, 0.42146419,
       0.49967021, 0.58731842, 0.68415936, 0.7895391 , 0.90268701,
       1.02214436, 1.14585189, 1.27148913, 1.39612926, 1.51638416,
       1.62857532, 1.72903813, 1.81452862, 1.88275929, 1.93293148,
       1.96611102, 1.9851947 , 1.9943386 , 1.9978019 , 1.99865198,
       1.99878377])
```

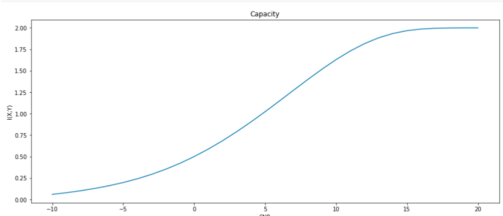
```
pbbs = 0.5 - paas
pbbs
```

```
array([0.34374913, 0.34394031, 0.34430381, 0.34502169, 0.34721558,
       0.31479596, 0.18600793, 0.07272666, 0.06745601, 0.06662427,
       0.06719935, 0.06829508, 0.0706318 , 0.0750943 , 0.08588186,
       0.10362774, 0.11530987, 0.12853202, 0.14215296, 0.15695149,
       0.17259394, 0.18765018, 0.20173559, 0.21534811, 0.22707176,
       0.23703831, 0.24409633, 0.24835733, 0.25025045, 0.25060684,
       0.25082245])
```

```
bbs = np.sqrt((0.5-(aas**2)*paas)/(0.5-paas))
bbs
```

```
array([1.00034933, 1.00039625, 1.00048515, 1.00065951, 0.99984329,
       1.25966941, 1.60438974, 2.04965362, 2.12353912, 2.13380803,
       2.12740515, 2.11534522, 2.08888796, 2.04322201, 1.96778473,
       1.86213859, 1.79790168, 1.73291147, 1.67202513, 1.6129768 ,
       1.55725442, 1.50815848, 1.46563803, 1.42822134, 1.39782165,
       1.37398162, 1.35756619, 1.34746715, 1.34316465, 1.32910171,
       1.30926037])
```

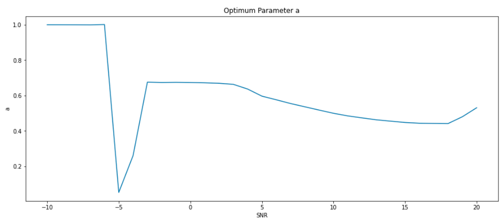
```
plt.figure(figsize=(15,6))
plt.plot(k,Hs)
plt.title("Capacity")
plt.xlabel("SNR")
plt.ylabel("I(X;Y)")
plt.show()
```



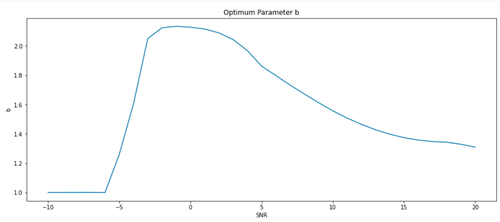
Comments: the upper bound of the capacity is 2, as expected if there is no noise. The capacity is always increasing.

(b) Plot of respective optimum parameters as function of σ^2

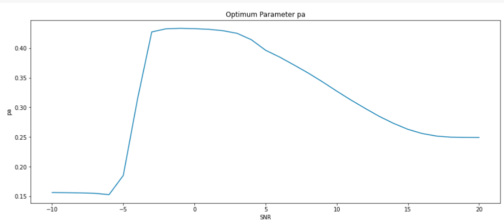
```
plt.figure(figsize=(15,6))
plt.plot(k,aas)
plt.title("Optimum Parameter a")
plt.xlabel("SNR")
plt.ylabel("a")
plt.show()
```



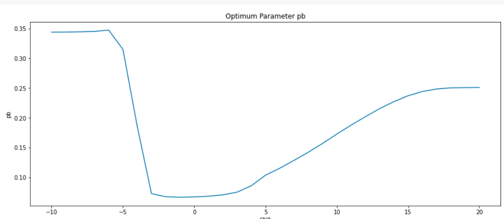

```
plt.figure(figsize=(15,6))
plt.plot(k,bbs)
plt.title("Optimum Parameter b")
plt.xlabel("SNR")
plt.ylabel("b")
plt.show()
```




```
plt.figure(figsize=(15,6))
plt.plot(k,paas)
plt.title("Optimum Parameter pa")
plt.xlabel("SNR")
plt.ylabel("pa")
plt.show()
```




```
plt.figure(figsize=(15,6))
plt.plot(k,pbbs)
plt.title("Optimum Parameter pb")
plt.xlabel("SNR")
plt.ylabel("pb")
plt.show()
```



Analysis:

- Parameter a and b tend to gather at 1 when SNR is large negative number while lowering p_a to prevent ambiguity between a and b. In other words, it kills the source a and separates $-b$ from b using literally all its power. Since p_a and a is very small, the power constraint prevents $-b$ and b from separating too far (*i.e.*, -1 to 1).
- There is an interesting event starting at SNR $= -5$, the system doesn't kill a, but it decided that there is a better option. Because the noise has calmed down a little, it now utilizes a. Since source a is low-powered, it also enables $-b$ and b to separate even farther. We get more channel (although it also means more noise), but it is more

preferable now since the noise is weaker.

- Parameter p_a and p_b tend to be uniformly distributed (*i.e.*, 0.25) when SNR is large positive number. This is done to maximize entropy of the sources while keeping maximum allowed distance from each other to prevent ambiguity.