



بسمه تعالی

دانشگاه صنعتی شریف

دانشکده مهندسی برق

سیستم های مخابراتی - دکتر پاکروان - زمستان ۱۴۰۲

پروژه

پارسا حاتمی ۴۰۰۱۰۰۹۶۲

## فهرست مطالب

۳	۱	مقدمه
۳	۲	پیاده سازی بلوک ها به صورت مجزا
۳	۱.۲	Divide,Combine
۴	۲.۲	Pulse Shaping
۴	۳.۲	AnalogMod
۵	۴.۲	Channel
۷	۵.۲	AnalogDemod
۷	۶.۲	MatchedFilter
۸	۳	انتقال دنباله تصادفی و
۸	۱.۳	PAM
۸	۱.۱.۳	
۱۳	۲.۱.۳	
۱۳	۳.۱.۳	
۱۴	۲.۳	PSK
۱۴	۱.۲.۳	
۱۹	۲.۲.۳	
۱۹	۳.۲.۳	
۲۰	۲.۳	FSK
۲۰	۱.۳.۳	
۲۰	۲.۳.۳	
۲۵	۳.۳.۳	
۲۵	۴.۳.۳	
۲۶	۴.۳	
۲۶	۴	انتقال دنباله ای از اعداد ۸ بیتی
۲۶	۱.۴	SourceGenerator,OutputDecoder
۲۷	۲.۴	Block output
۳۲	۳.۴	Error distribution for different variances
۳۳	۴.۴	Error distribution for infinity variance
۳۳	۵	فسرگستر
۳۳	۱.۵	
۳۴	۲.۵	
۳۵	۳.۵	
۳۶	۴.۵	
۳۷	۵.۵	
۳۷	۶.۵	
۳۸	۷.۵	
۴۰	۸.۵	
۴۱	۹.۵	
۴۲	۱۰.۵	
۴۴	۱۱.۵	

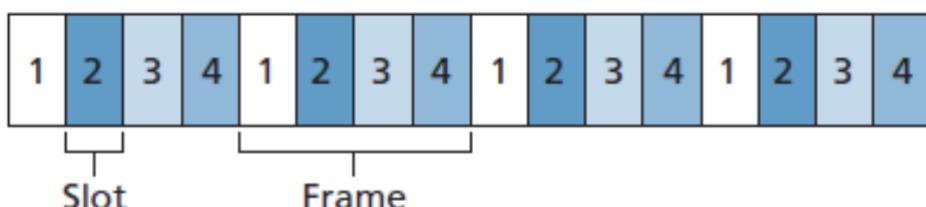
## ۱ مقدمه

در این پژوهه قصد داریم یک سیستم مخابرات دیجیتال را به صورت کامل شبیه سازی کنیم.

## ۲ پیاده سازی بلوک ها به صورت مجزا

### ۱.۲ Divide, Combine

توابع خواسته شده در فایل نوتبوک قابل مشاهده است. برای اینکه سیستم ما به یک سیستم real time نزدیکتر شود ما باید یک بیت از رشته اول و یک بیت از رشته دوم را به صورت متناوب به دنباله هم قرار دهیم تا هردو پیام با یکدیگر به صورت همزمان و موازی ارسال شوند و ظرفیت زمانی کانال بین دو کاربر به صورت مساوی تقسیم شود به عبارتی بیت های ما در رشته اصلی به صورت یکی در میان مربوط به رشته اول و دوم میشود. شکلی برای فهم بهتر برای تقسیم کردن رشته اصلی به ۴ رشته:



شکل ۱: تقسیم دنباله ورودی

خروجی تابع مورد نظر برای یک مثال رندوم:

```

def Divide(sequence):
    sequence1 = [int(sequence[i]) for i in range(0, len(sequence), 2)]
    sequence2 = [int(sequence[i]) for i in range(1, len(sequence), 2)]
    return sequence1, sequence2

def Combine(sequence1, sequence2):
    sequence = [int(val) for pair in zip(sequence1, sequence2) for val in pair]
    return sequence
✓ 0.0s

sequence = [0,1,0,1,0,0,1,1,0,1,1,1,0,0,0,1]
sequence1, sequence2 = Divide(sequence)
combined_sequence = Combine(sequence1, sequence2)

print("Sequence 1:", sequence1)
print("Sequence 2:", sequence2)
print("Combined Sequence:", combined_sequence)

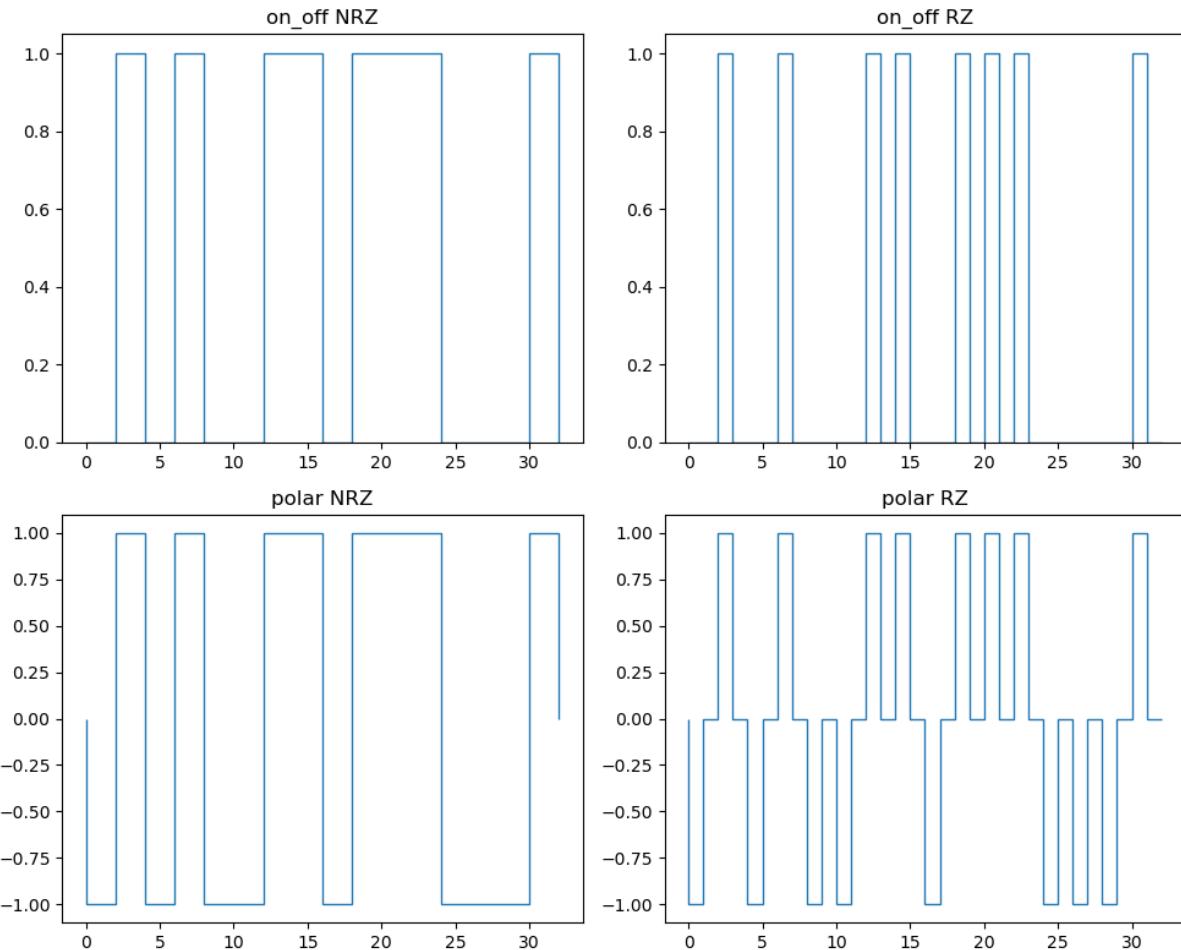
✓ 0.0s
Sequence 1: [0, 0, 0, 1, 0, 1, 0, 0]
Sequence 2: [1, 1, 0, 1, 1, 1, 0, 1]
Combined Sequence: [0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1]

```

شکل ۲: مثال برای بررسی کار کرد تابع

**Pulse Shaping ۲.۲**

تابع خواسته شده در فایل نوتبوک قابل مشاهده است همچنین خروجی آن برای ۴ حالت کدینگ مختلف نیز در زیر آورده شده است :

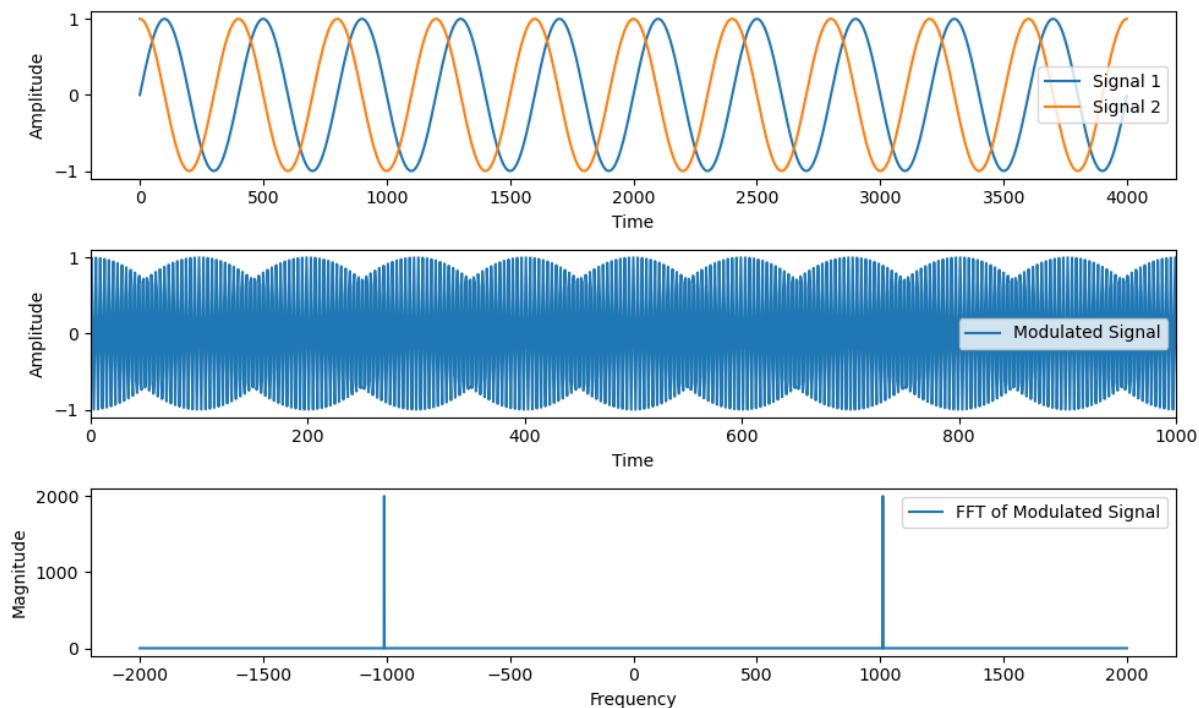


شکل ۳: مثال برای بررسی کار کرد تابع

در مثال آورده شده برای مشاهده راحت هر بیت را دو واحد زمانی فرض کرده ام (کمترین مقداری که میتوان هم با آن NRZ و هم RZ را نشان داد.

**AnalogMod ۳.۲**

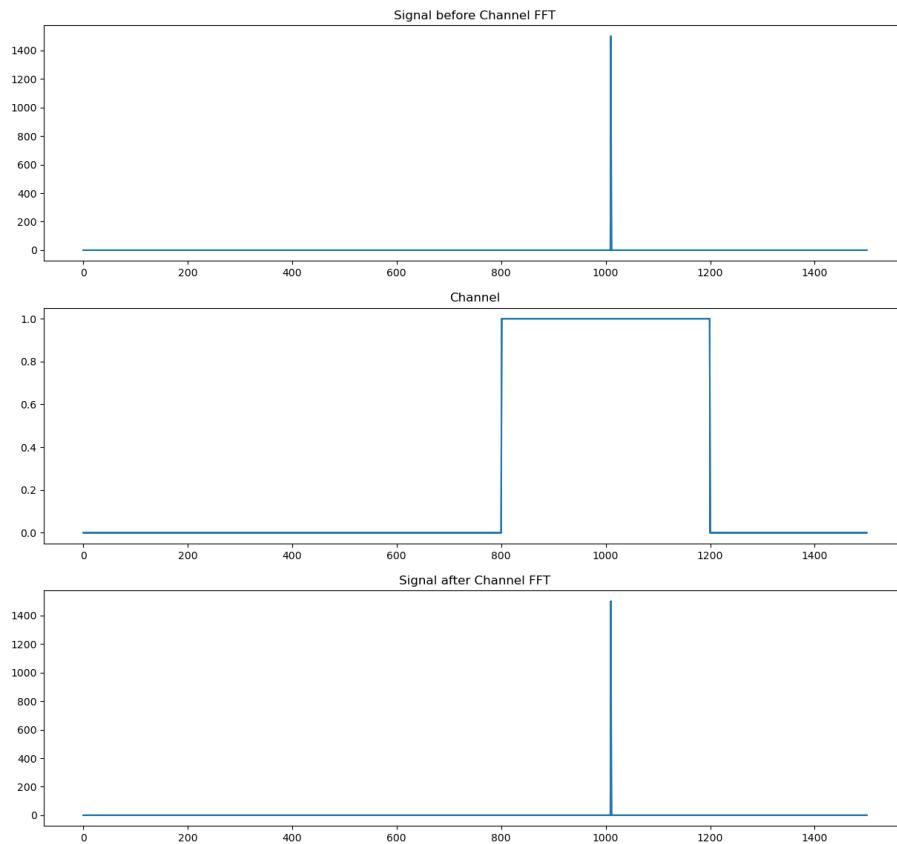
برای پیاده سازی این تابع مطابق بلوک دیاگرامی که در ابتدای پژوهه آمده است عمل شده است. یکی از سیگنال ها در کریر کسینوس و دیگری در کریر سینوس ضرب شده و با یکدیگر جمع شده اند. برای بررسی کار کرد این تابع به عنوان ورودی یک تابع سینوسی و یک تابع کسینوسی در نظر گرفته شده است و خروجی ها را رسم کرده ام که مطابق شکل زیر است و نشان میدهد که تابع به درستی عمل میکند :



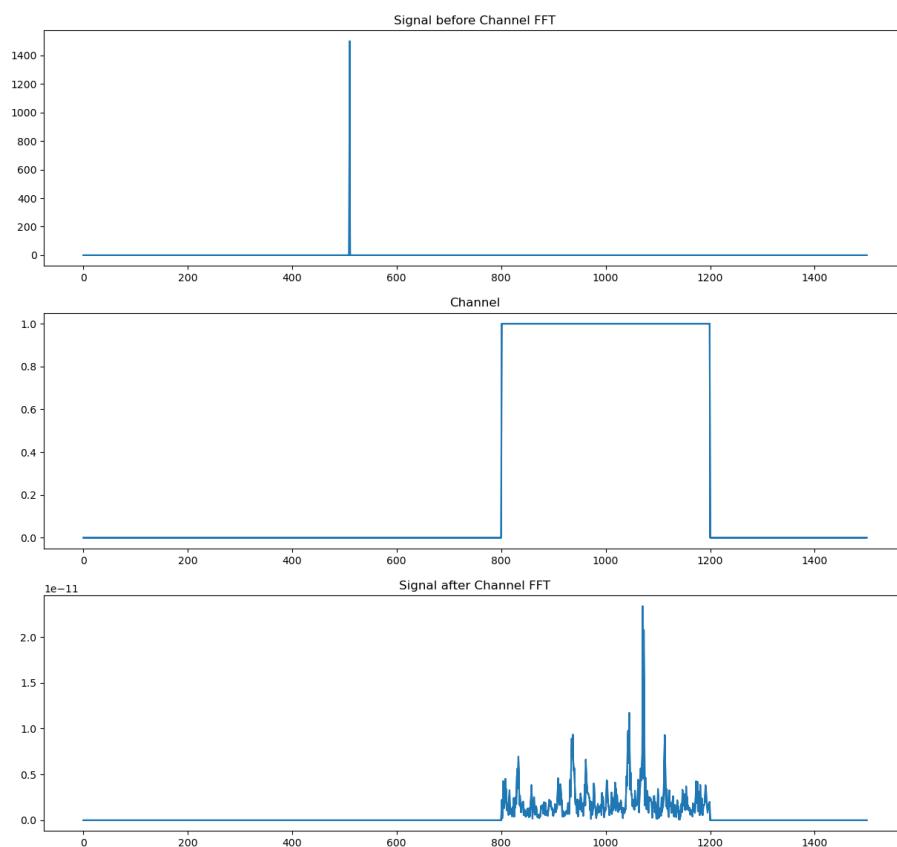
شکل ۴: مثال برای بررسی کار کرد تابع

## Channel ۴.۲

برای پیاده سازی این تابع کافی است در حوزه فرکانس یک فیلتر میانگذر بسازیم و تبدیل فوریه تابع را در آن ضرب کنیم و سپس از حاصل فوریه وارون بگیریم تا سیگنال حاصل پس از عبور از کانال بدست آید. برای نحوه طراحی فیلتر میانگذر نیز کافی است تابعی تعیین کنیم که در بین دو فرکانس مقدار ۱ و در باقی فرکانس ها مقدار ۰ لحاظ کند. نمونه خروجی تابع اثر کانال بر روی دو سیگنال که یکی در پهنهای باند کانال قرار دارد و دیگر خارج از این باند است. همانطور که از روی اندازه نمودار ها مشخص است سیگنالی که در پهنهای باند کانال قرار دارد به خوبی عبور میکند ولی سیگنالی که در خارج این پهنهای باند است عملا نابود میشود. با دامنه خیلی ناچیزی (در اردر ۱۰ به توان -۱۱) عبور میکند.



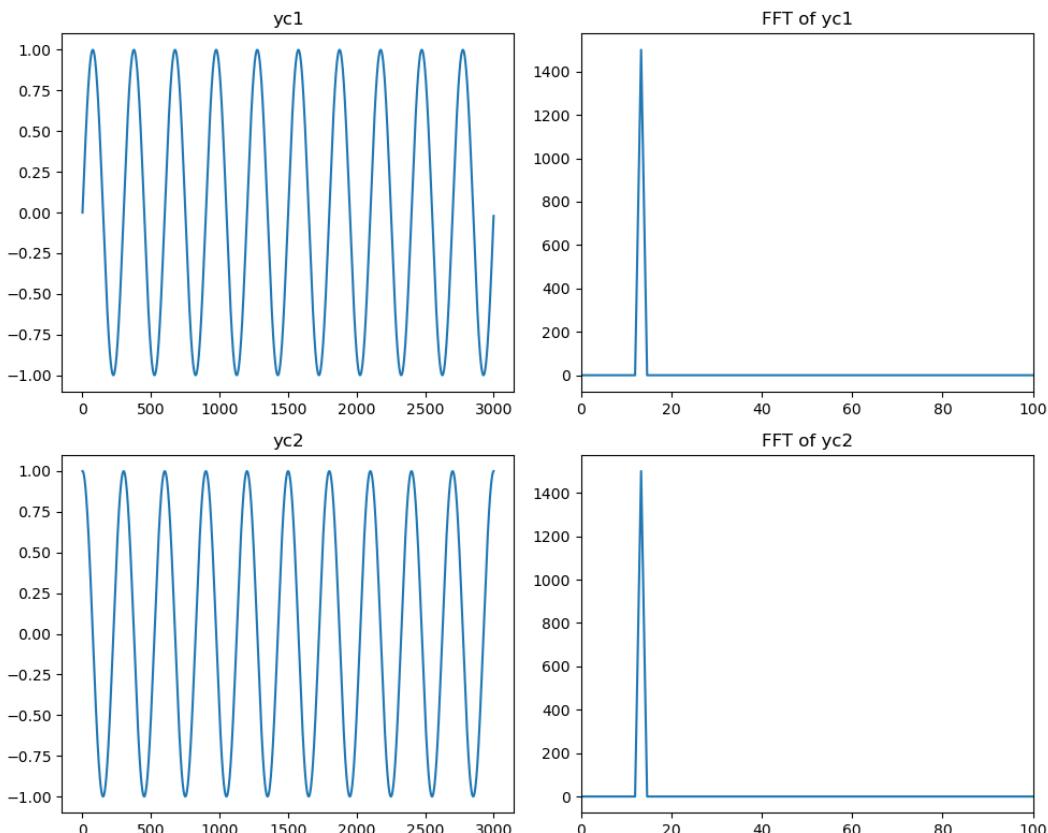
شکل ۵: قرارگیری در پهنهای باند کانال



شکل ۶: قرارگیری در خارج از پهنهای باند کانال

## 5.2 Analog Demod

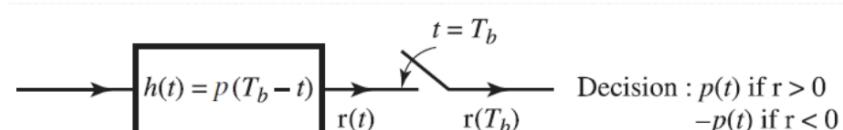
در این بخش هم مطابق بلوک دیاگرام ابتدای پروژه عمل شده است. به این صورت که سیگنال مخابره شده را در کسینوس و سینوس کریم ضرب میکنیم و سپس از یک فیلتر پایین گذر عبور میدهیم و در نهایت گین مورد نظر را قرار میدهیم تا به دامنه سیگنال اولیه دست پیدا کنیم. نمونه خروجی تابع برای ورودی سینوسی و کسینوسی با فرکانس  $10$  هرتز:



شکل ۷: مثال برای کار کرد تابع

## 6.2 Matched Filter

طبق گفته دستور پروژه برای طراحی این بخش مطابق اسلاید های درس عمل شده است که در زیر اسلاید مربوطه آورده شده است:



شکل ۸: Matched Filter

مطابق بلوک دیاگرام آورده شده است در این بلوک باید سیگنال با فلیپ و شیفت خورده خود کانوالو شود. تابع پیاده سازی شده طبق این منطق به صورت زیر است:

```

def MatchedFilt(signal, pulse_one, pulse_zero):
    filter_one = np.flip(pulse_one)
    filter_zero = np.flip(pulse_zero)

    filter_one_fft = np.fft.fft(filter_one)
    filter_zero_fft = np.fft.fft(filter_zero)

    signal_one = np.zeros_like(signal, dtype=np.complex128)
    signal_zero = np.zeros_like(signal, dtype=np.complex128)

    pulse_length = len(pulse_one)
    for i in range(0, len(signal), pulse_length):
        signal_fft = np.fft.fft(signal[i:i + pulse_length])
        signal_one[i:i + pulse_length] = np.fft.ifft(signal_fft * filter_one_fft)
        signal_zero[i:i + pulse_length] = np.fft.ifft(signal_fft * filter_zero_fft)

    matched_filter_one = signal_one[pulse_length - 1::pulse_length]
    matched_filter_zero = signal_zero[pulse_length - 1::pulse_length]

    detected_bits = np.array([1 if matched_filter_one[i] > matched_filter_zero[i] else 0 for i in range(len(matched_filter_one))])
    return matched_filter_one, matched_filter_zero, detected_bits

```

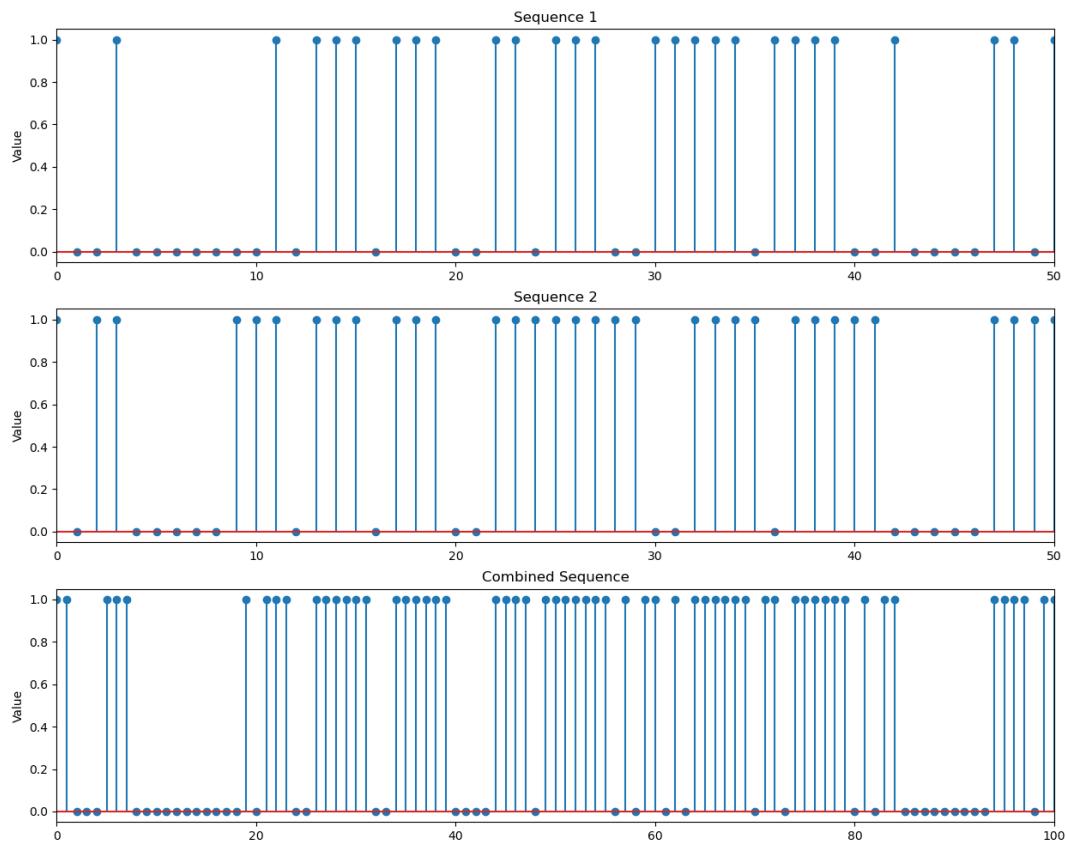
شکل ۹: Matched Filter

### ۳ انتقال دنباله تصادفی ۰ و ۱

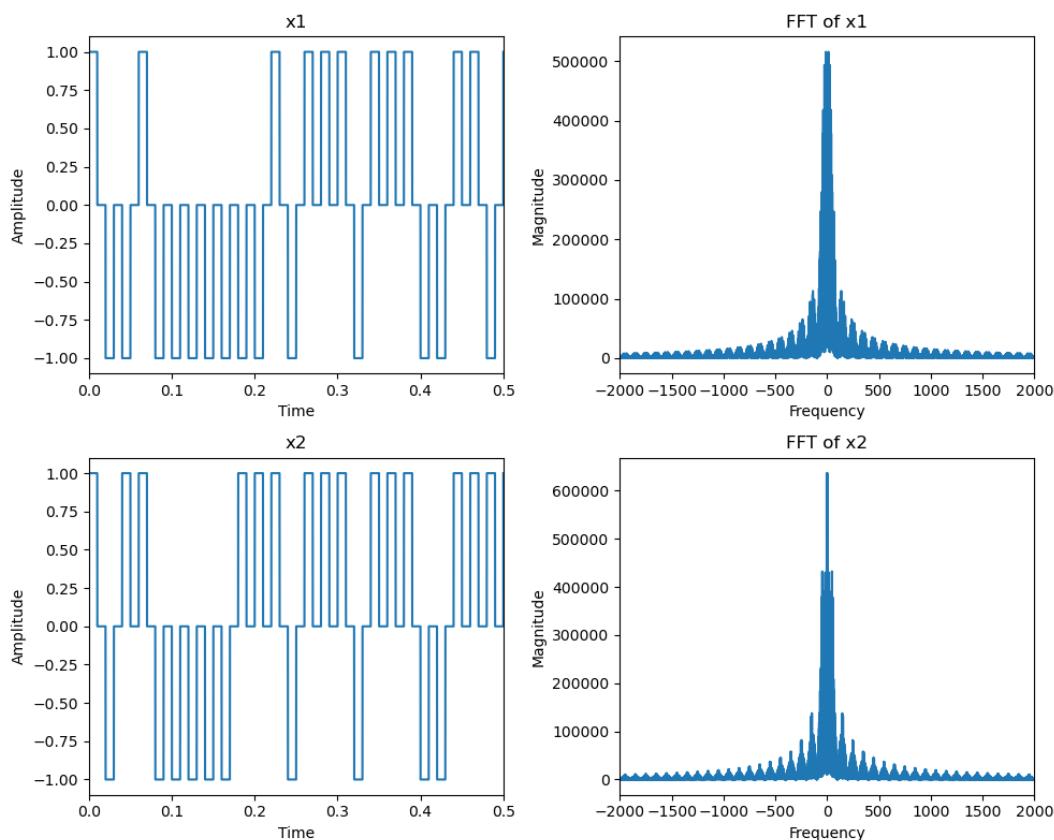
PAM ۱.۳

۱.۱.۳

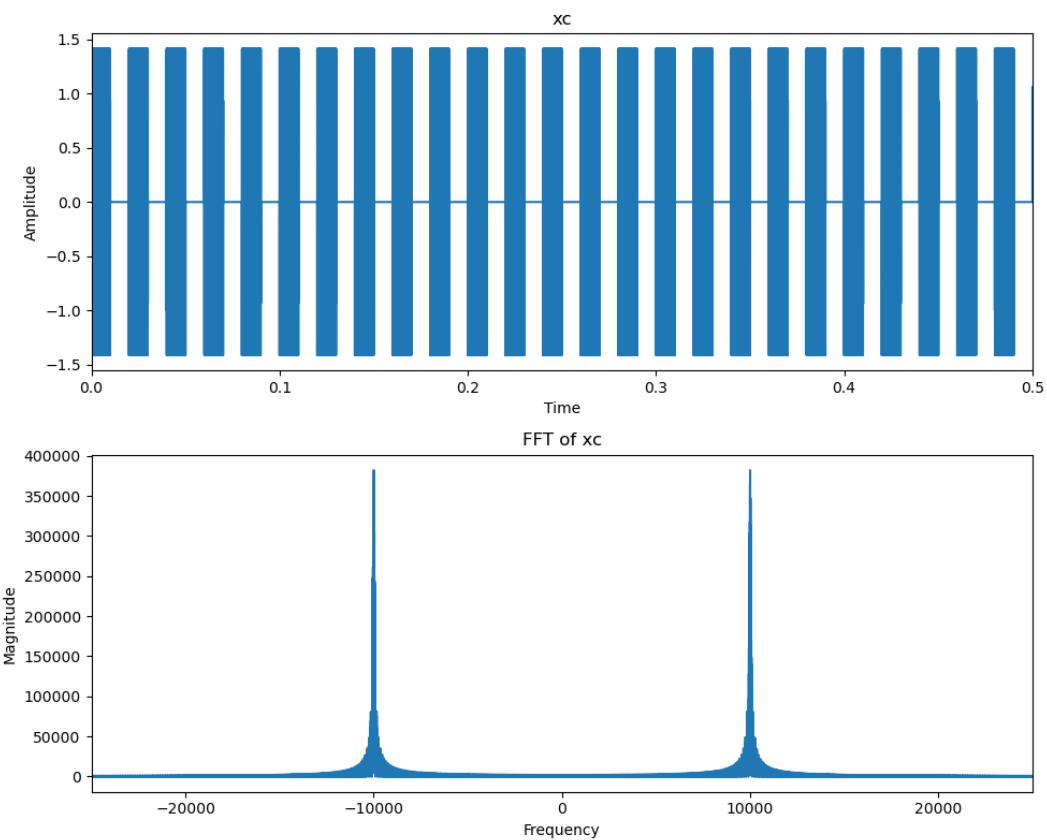
برای خروجی تابع Divider محدوده محور افقی را روی ۵۰ محدود کرده ام تا شکل خروجی قشنگتر دیده شود و گرنه دنباله به اندازه کافی بلند در نظر گرفته شده است. همچنین لازم به ذکر است که در این بخش طول هر بیت دارای ۱۰۰۰۰ سمپل است چون هر ثانیه دارای ۱۰۰۰۰۰۰ سمپل است و طول هر بیت نیز ۰.۰۱ ثانیه است پس هر بیت شامل ۱۰۰۰۰ سمپل میشود. خروجی Divider



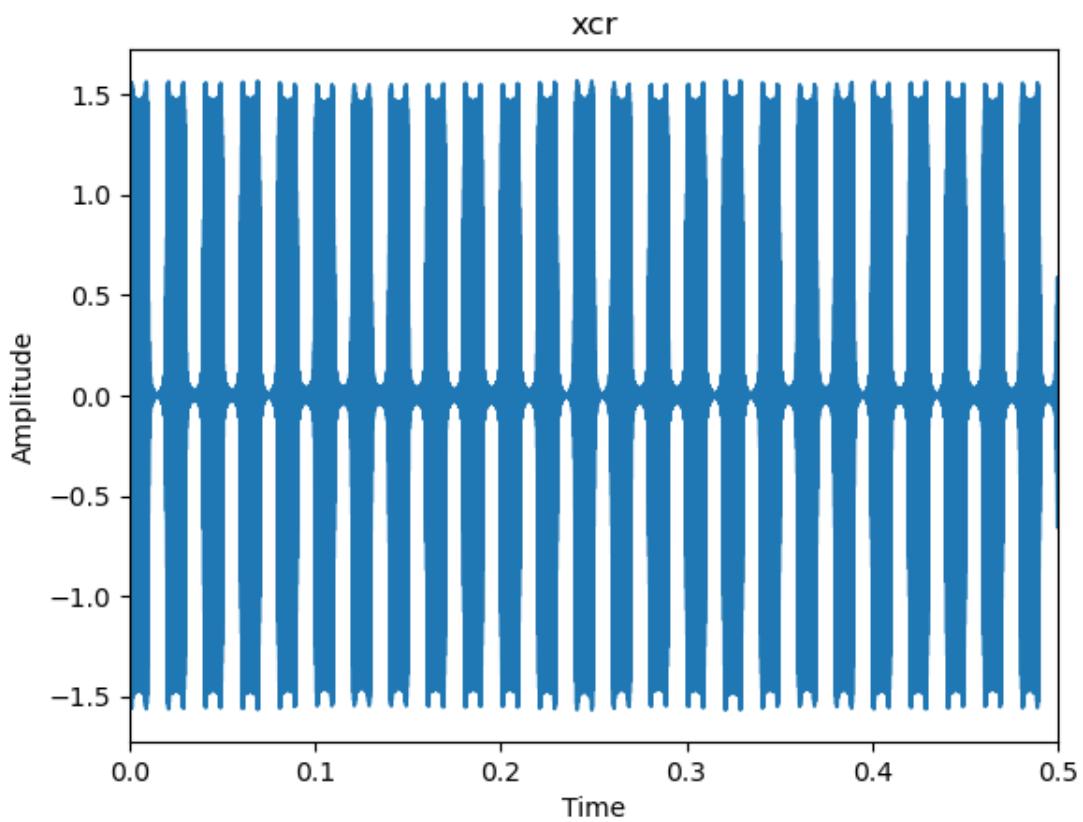
شکل ۱۰: خروجی بلوک Divider



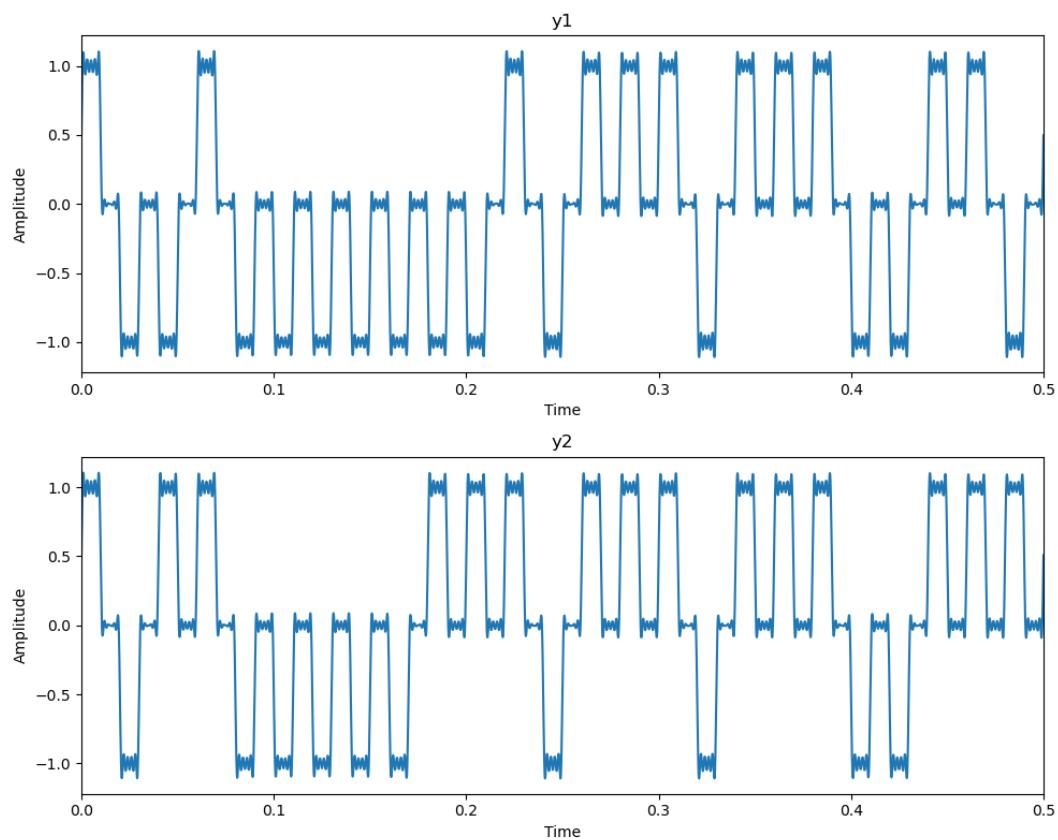
شکل ۱۱: خروجی بلوک Pulse Shaping



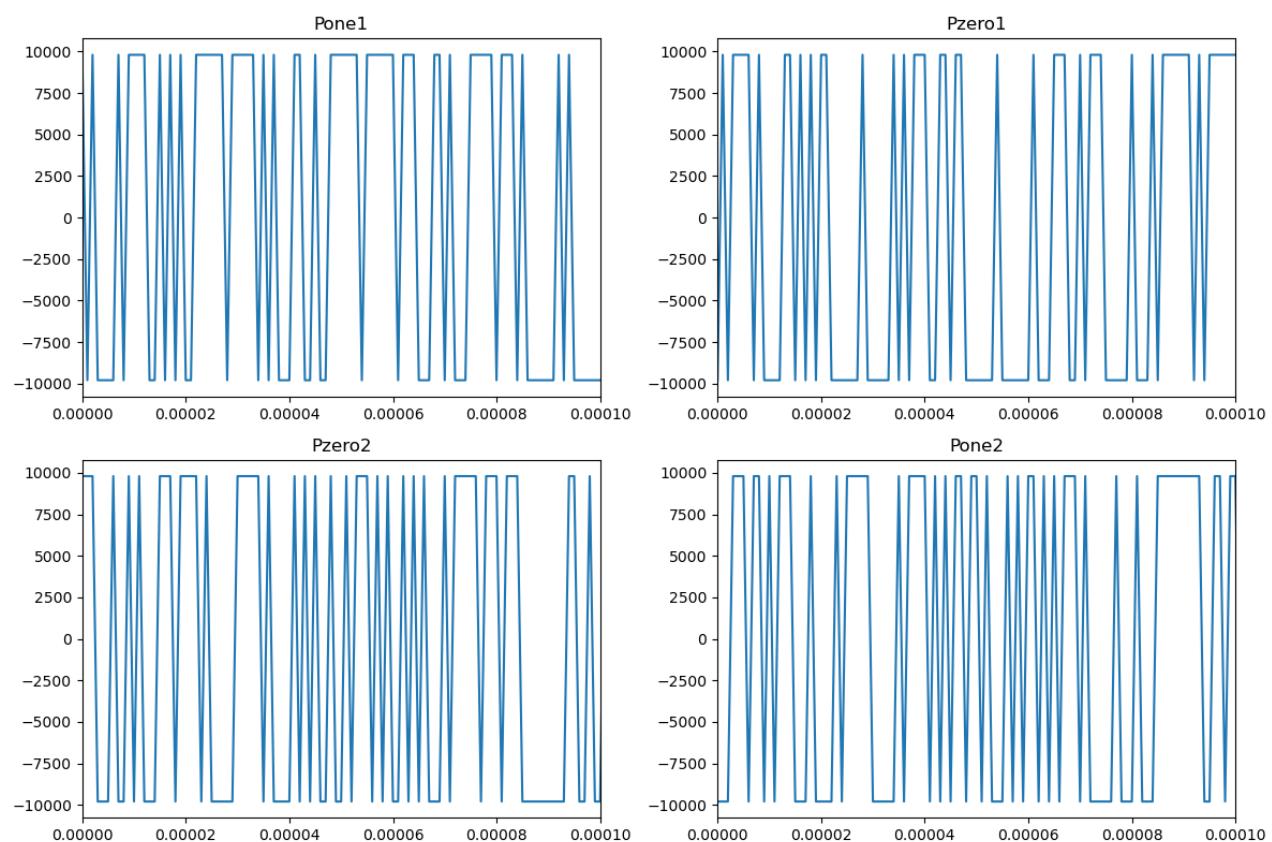
شکل ۱۲: خروجی بلوک AnalogMod



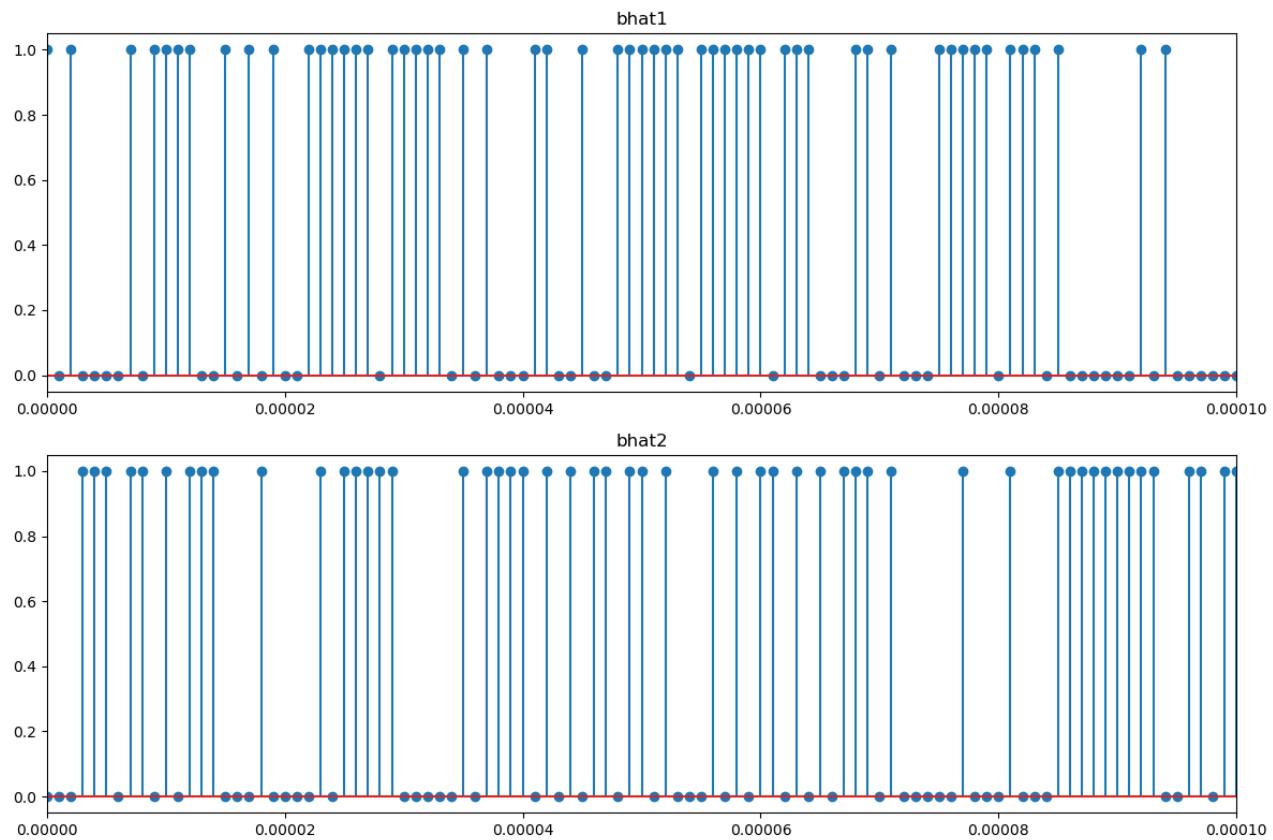
شکل ۱۳: خروجی بلوک Channel



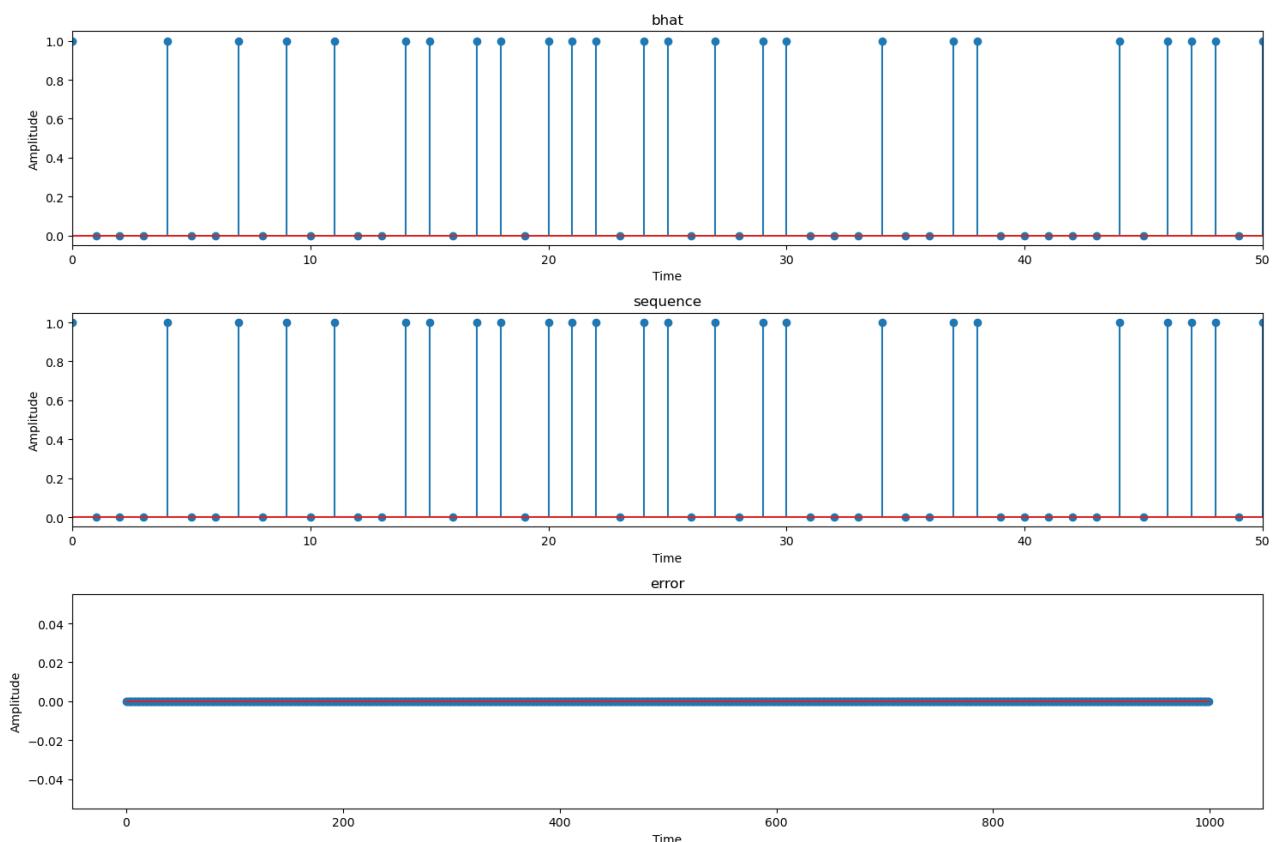
شکل ۱۴: خروجی بلوک AnalogDemod



شکل ۱۵: خروجی بلوک Matched Filter



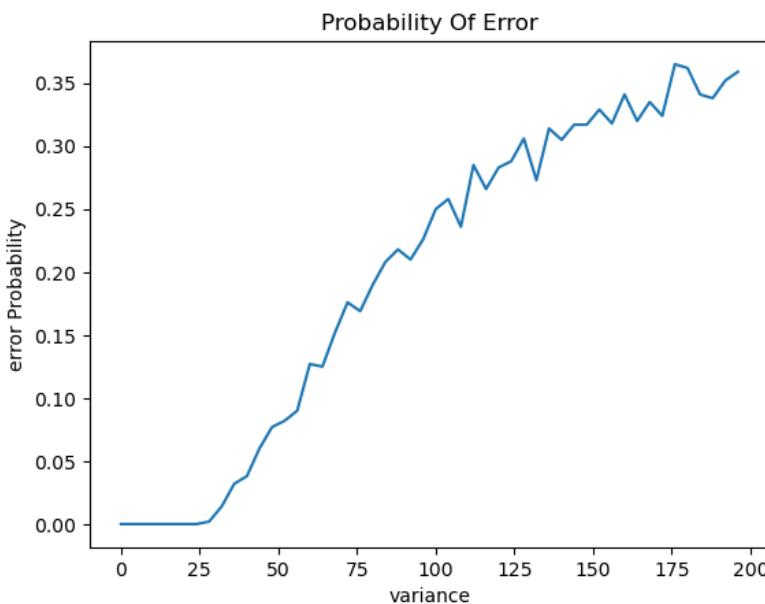
شکل ۱۶: خروجی بلوک Matched Filter



شکل ۱۷: خروجی بلوک Combine: سیگنال بازیابی شده

## ۲.۱.۳

حال در بخش بعدی نویز سفید را به سیگنال دریافت شده از کانال اضافه میکنیم و از بخش AnalogDemod به بعد مراحل را برای سیگنال نویزی دوباره طی میکنیم. این مراحل را تا واریانس ۲۰۰ با استپ های ۴ تایی طی میکنیم و سیگنال بازیابی شده را با سیگنال اولیه مقایسه میکنیم و خطاهای را بدست می آوریم. نمودار احتمال خطای بر حسب مقدار واریانس به صورت زیر میشود :

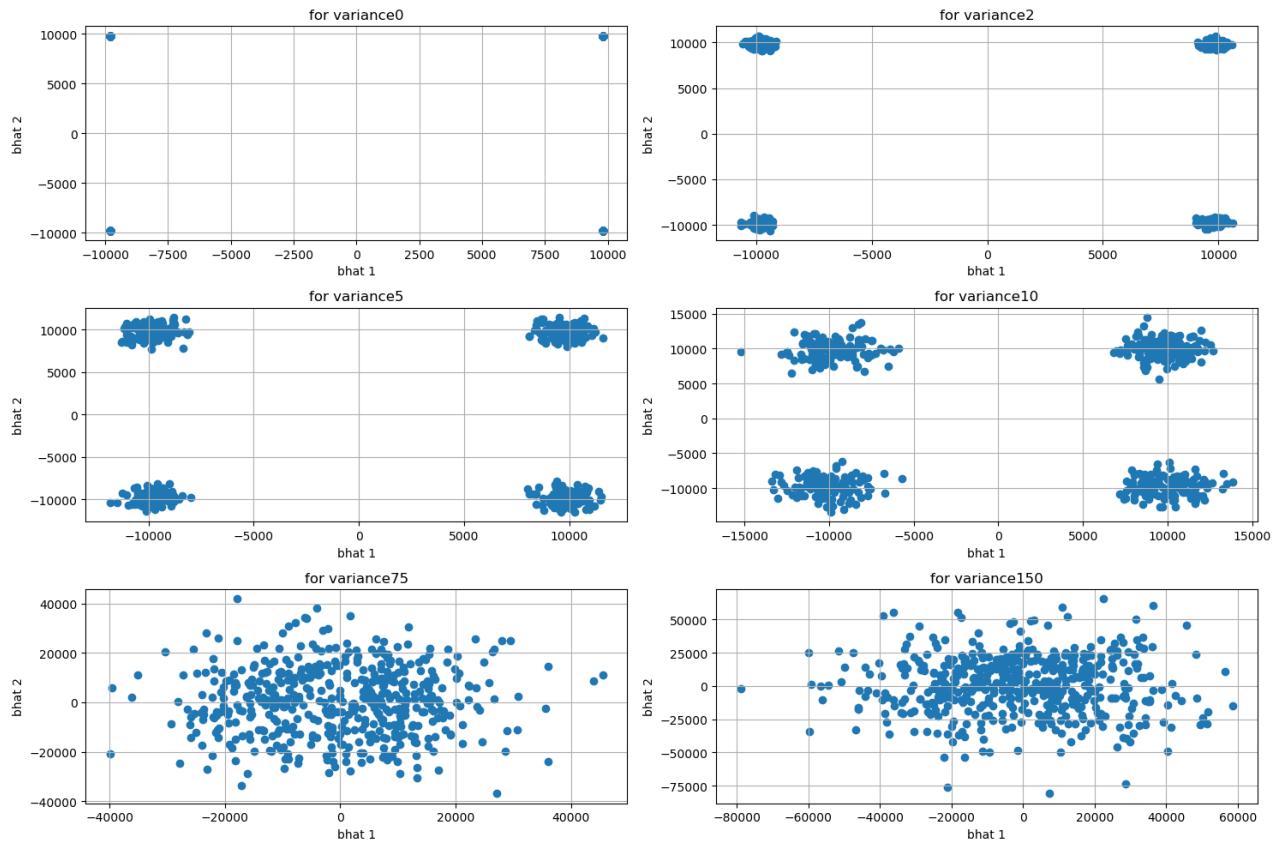


شکل ۱۸: نمودار احتمال خطای بر حسب واریانس

همانطور که از پیش نیز انتظار داشتیم و بدیهی نیز است هر چه واریانس نویز بیشتر باشد و مقادیر نویز دورتر از مقدار ۰ بتوانند قرار گیرند احتمال رخداد خطای و شناسایی اشتباه بیت برای ما بیشتر است. که نمودار به دست آمده نیز همین موضوع را تصدیق میکند. هر چه واریانس را افزایش میدهیم احتمال خطای در تشخیص بیت بیشتر میشود به صورتی که در حدود ۲۰۰ واریانس احتمال خطای نزدیک به ۰.۳۵ می رسد. البته بهتر بود که محور افقی از این نیز بزرگتر در نظر گرفته شود و همچنین استپ ها کوچکتر در نظر گرفته شود ولی به دلیل زمان زیادی که این بخش برای ران شدن نیاز داشت امکان پذیر نبود.

## ۳.۱.۳

حال در بخش بعدی برای ۶ مقدار واریانس که در بالای نمودارها قابل مشاهده است نمودار دو بعدی Signal Constellation را رسم میکنیم که به صورت زیر است :

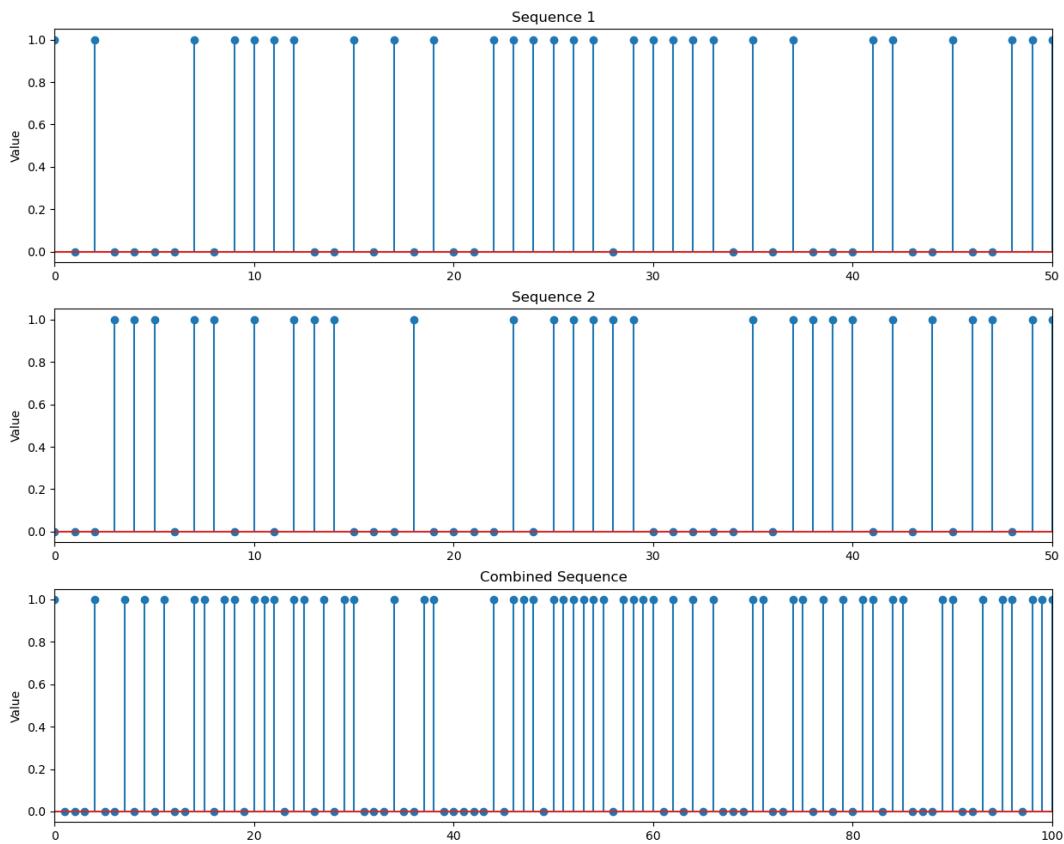


Signal Constellation : ۱۹

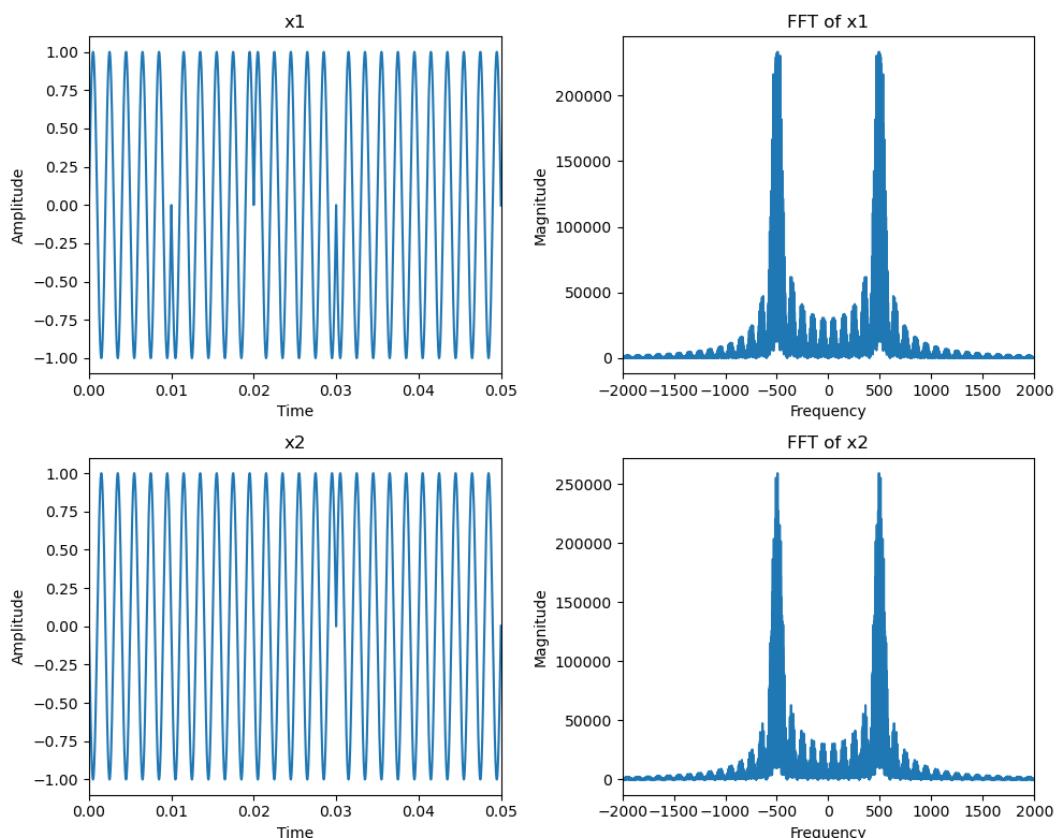
PSK ۲.۳

۱.۲.۳

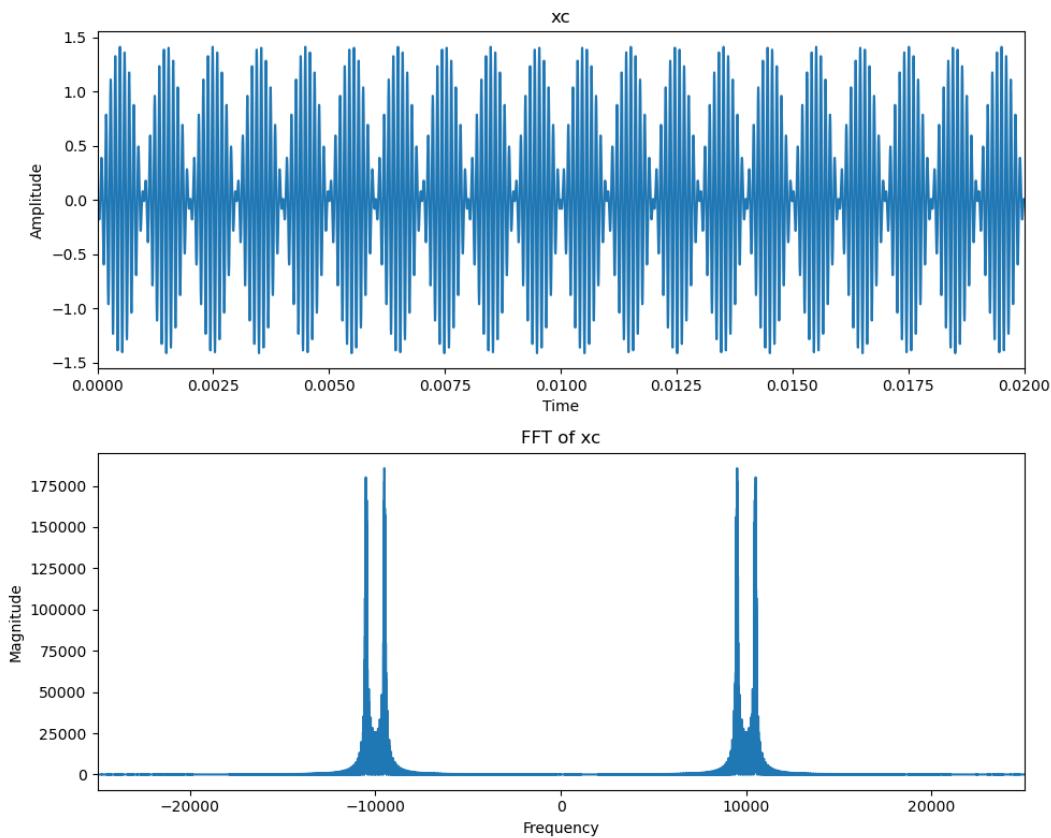
در این بخش تنها کافی است نوع پالس های صفر و یک تعریف شده را تغییر دهیم و مراحلی که برای بخش قبلی انجام دادیم را با پالس جدید تکرار کنیم.  
خروجی بلوک ها برای PSK :



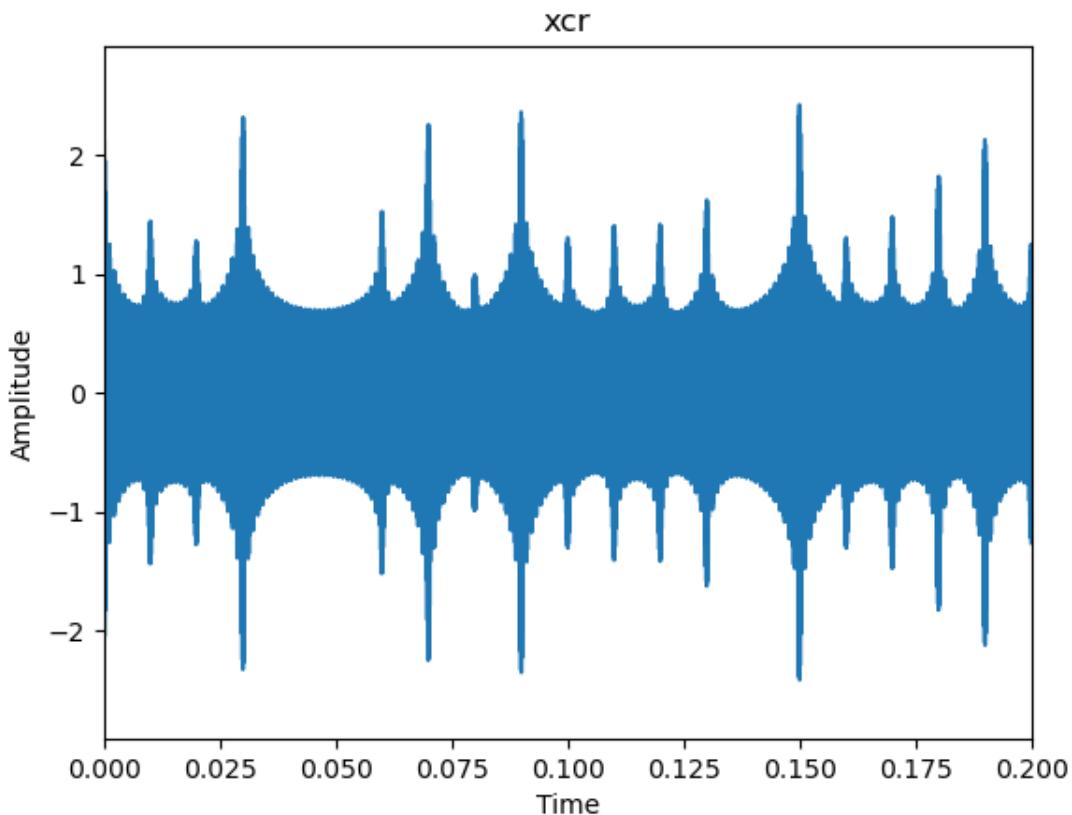
شکل ۲۰: خروجی بلوک Divider



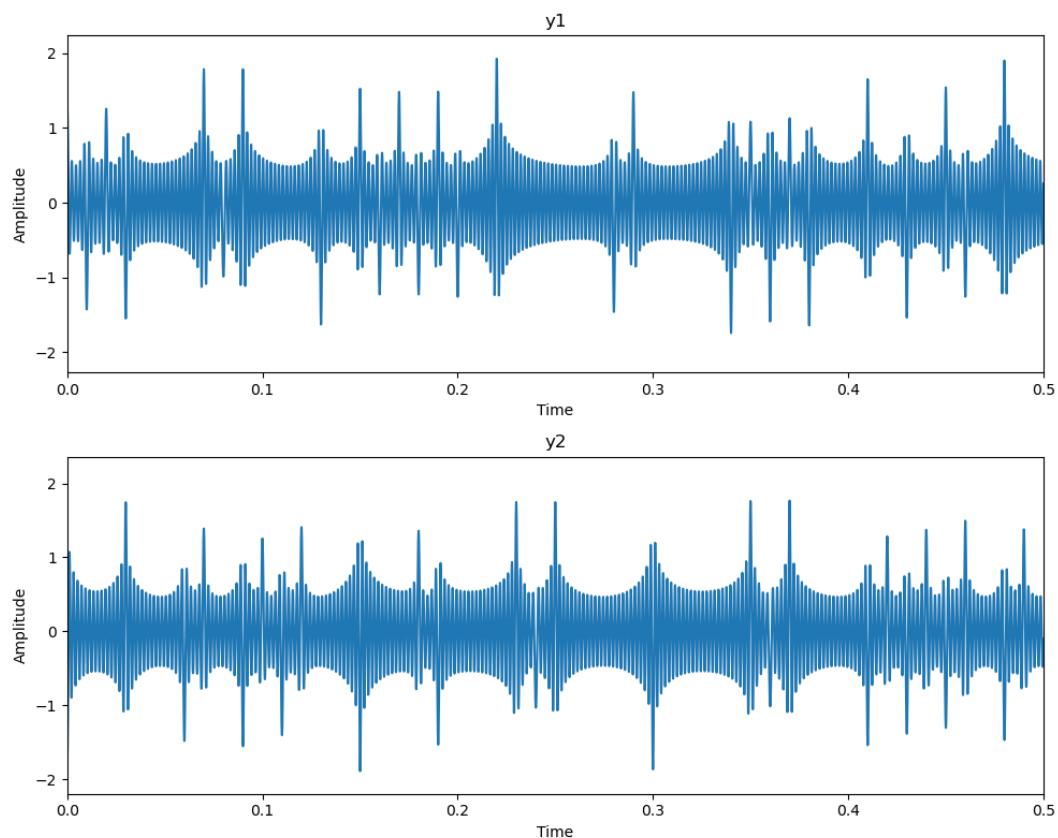
شکل ۲۱: خروجی بلوک Pulse Shaping



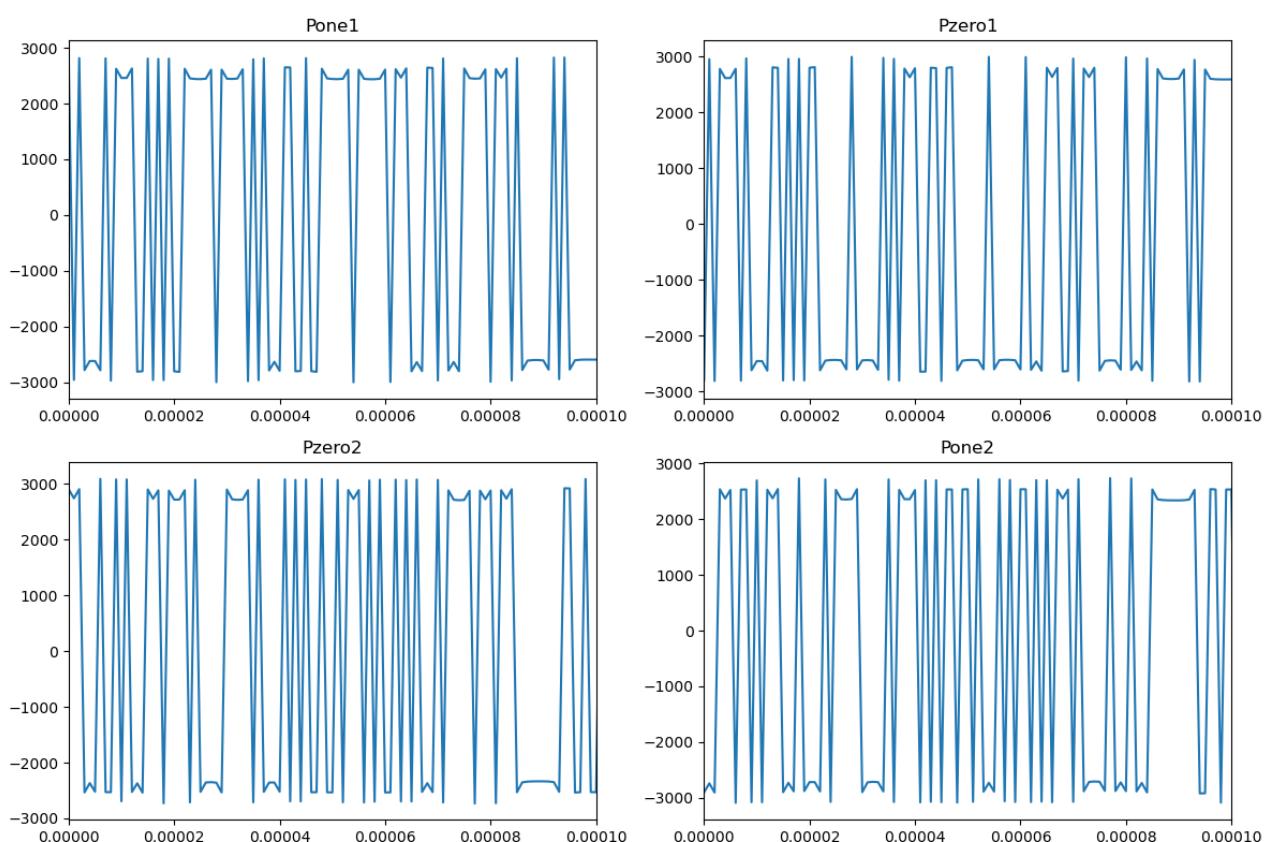
شکل ۲۲: خروجی بلوک AnalogMod



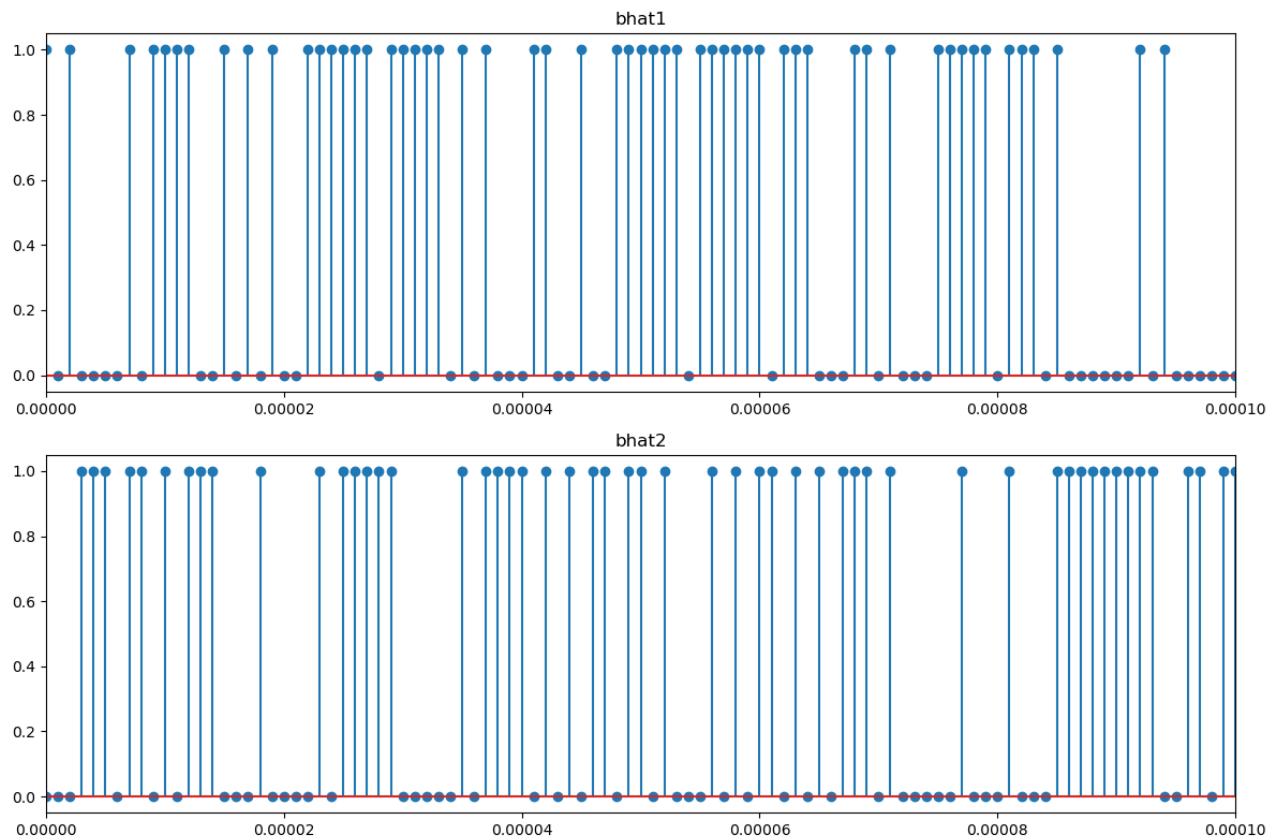
شکل ۲۳: خروجی بلوک Channel



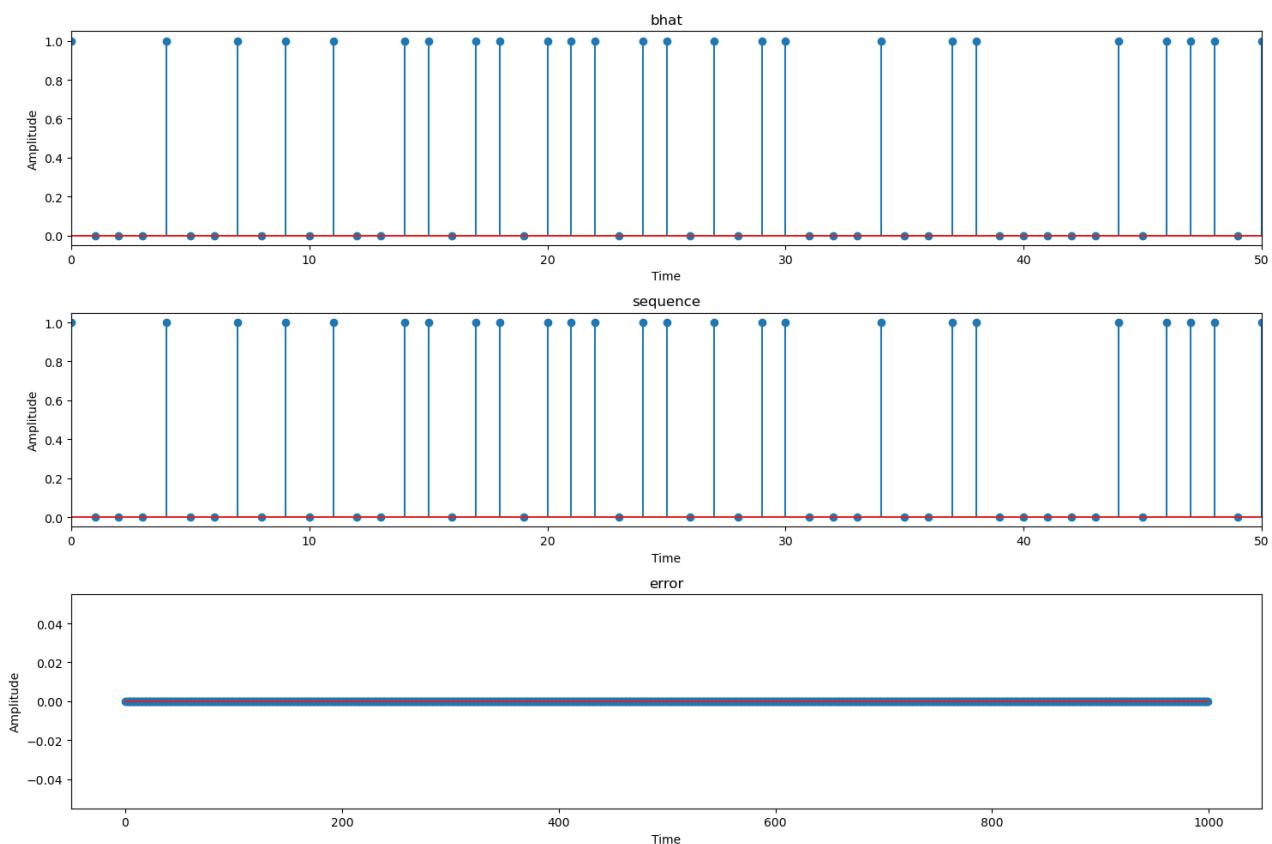
شکل ۲۴: خروجی بلوک AnalogDemod



شکل ۲۵: خروجی بلوک Matched Filter

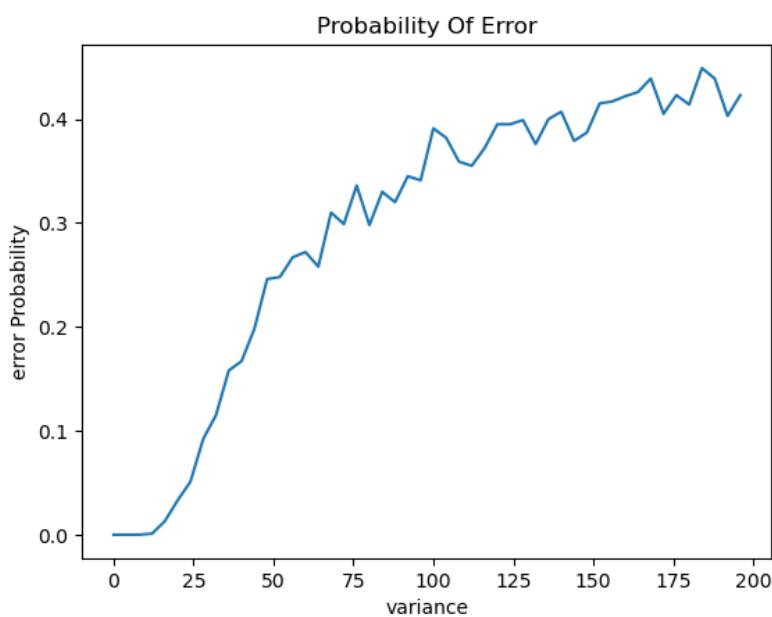


شکل ۲۶: خروجی بلوک Matched Filter



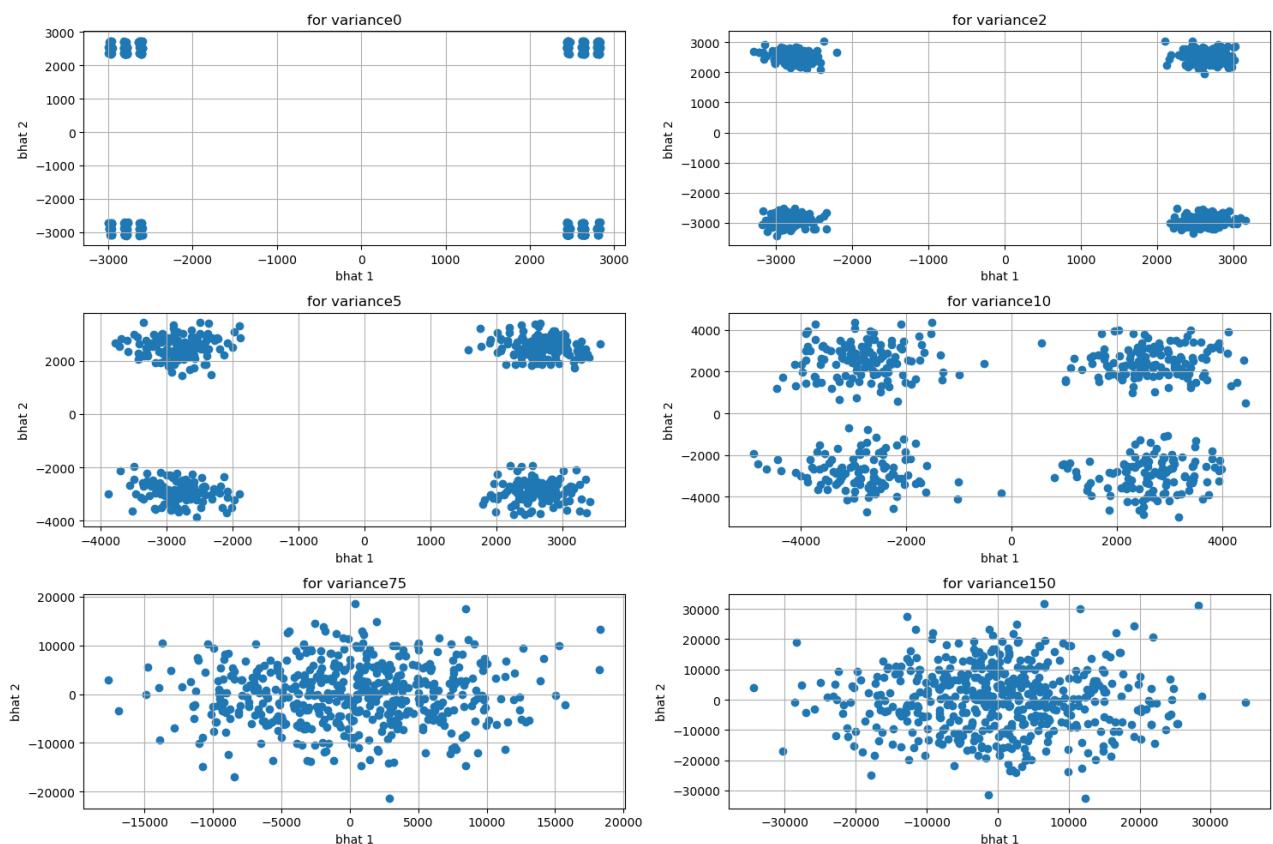
شکل ۲۷: خروجی بلوک Combine: سیگنال بازیابی شده

۲.۲.۳



شکل ۲۸: نمودار احتمال خطأ بر حسب واریانس

۳.۲.۳



شکل ۲۹: Signal Constellation

## FSK ۳.۳

## ۱.۳.۳

طبق مطالب ارائه شده در درس و اسلاید ها میدانیم تعامد به صورت زیر تعریف می شود :

$$\begin{aligned} \int_0^{T_s} A \sin(2\pi f_m t) A \sin(2\pi f_n t) dt &= 0 \quad m \neq n \\ \int_0^{T_s} A \sin(2\pi f_m t) A \sin(2\pi f_n t) dt \\ &= \frac{-A^2}{2} \int_0^{T_s} [\cos(2\pi(f_m + f_n)t) - \cos(2\pi(f_m - f_n)t)] dt \\ &= \frac{-A^2}{2} T_s \frac{\sin(2\pi(f_m + f_n)T_s)}{2\pi(f_m + f_n)T_s} + \frac{A^2}{2} T_s \frac{\sin(2\pi(f_m - f_n)T_s)}{2\pi(f_m - f_n)T_s} \end{aligned}$$

در اینجا داریم :

$$A = 1, f_m = 1500, f_n = 1000 \quad (1)$$

پس خواهیم داشت :

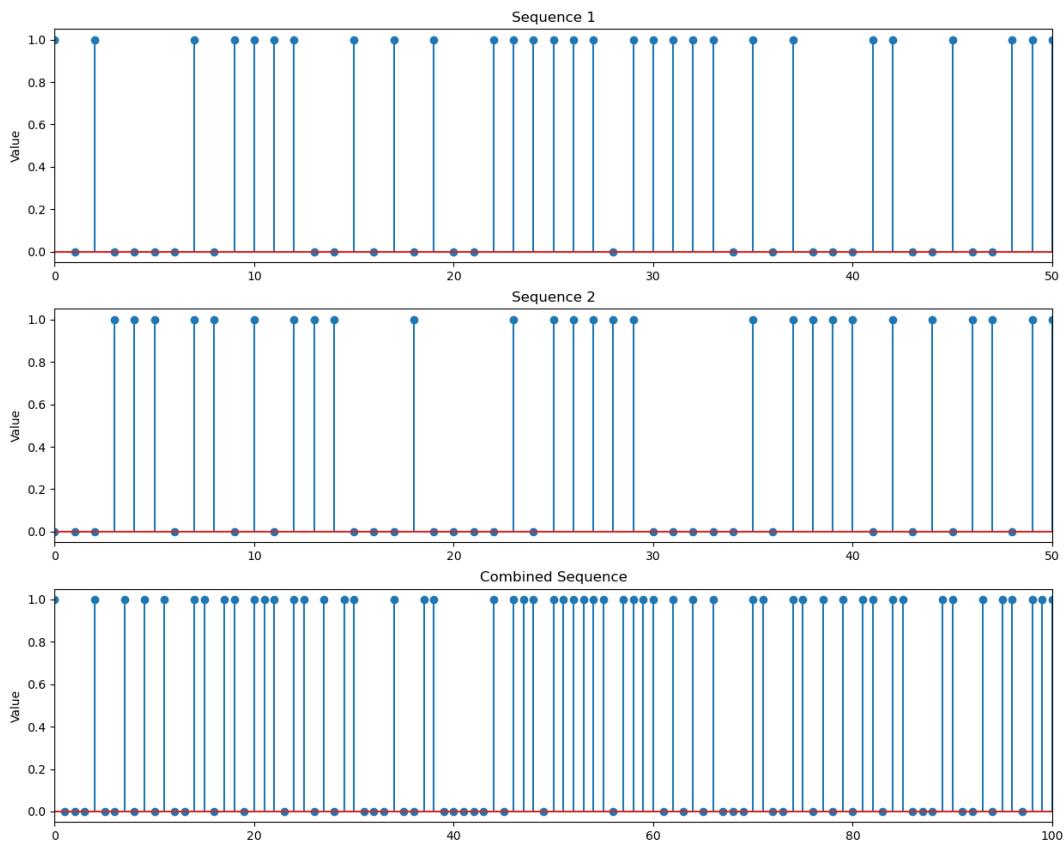
$$= \frac{-1}{2} \frac{\sin(2\pi(2500)T_s)}{2\pi(2500)} + \frac{1}{2} \frac{\sin(2\pi(500)T_s)}{2\pi(500)} \quad (2)$$

در هر دو کسر صورت بین منفی یک و یک است و مخرج برابر ۱۰۰۰ پی و ۵۰۰۰ پی است که در نتیجه هر دوی این کسر ها را میتوان برابر با صفر در نظر گرفت پس حاصل انتگرال مورد نظر صفر است و در نتیجه دو سیگنال بر هم متعامدند.

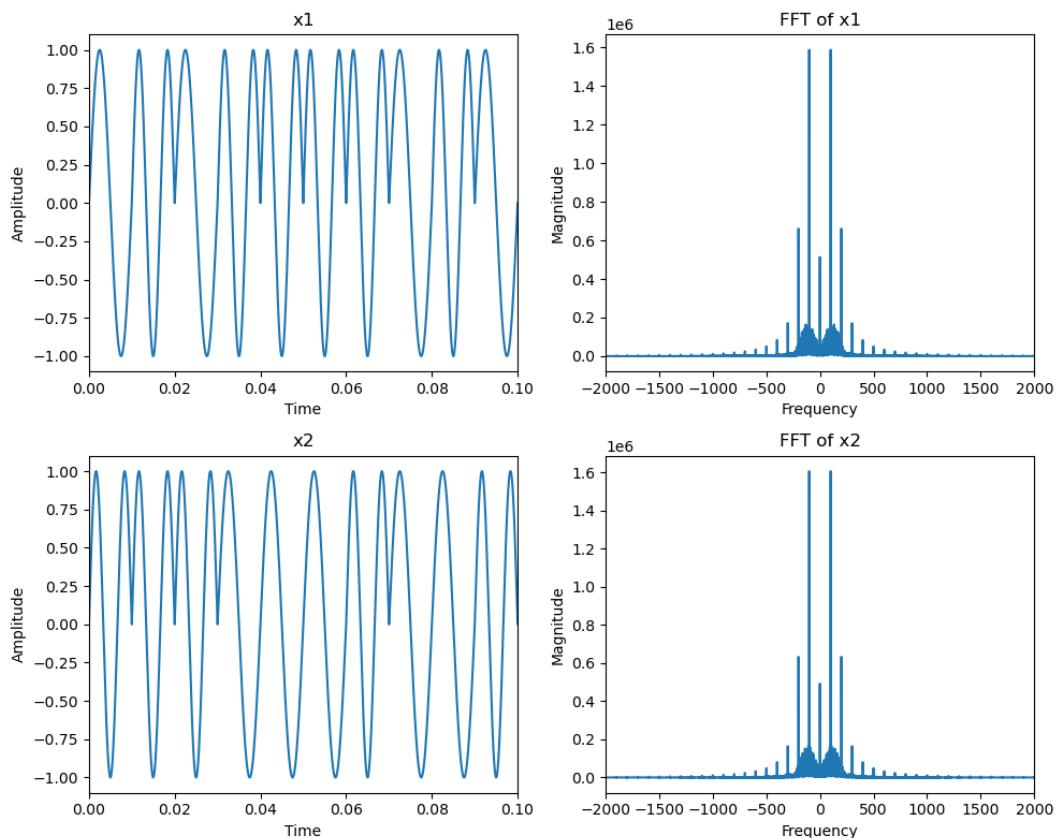
## ۲.۳.۳

در این بخش تنها کافی است نوع پالس های صفر و یک تعریف شده را تغییر دهیم و مراحلی که برای بخش های قبلی انجام دادیم را با پالس جدید تکرار کنیم.

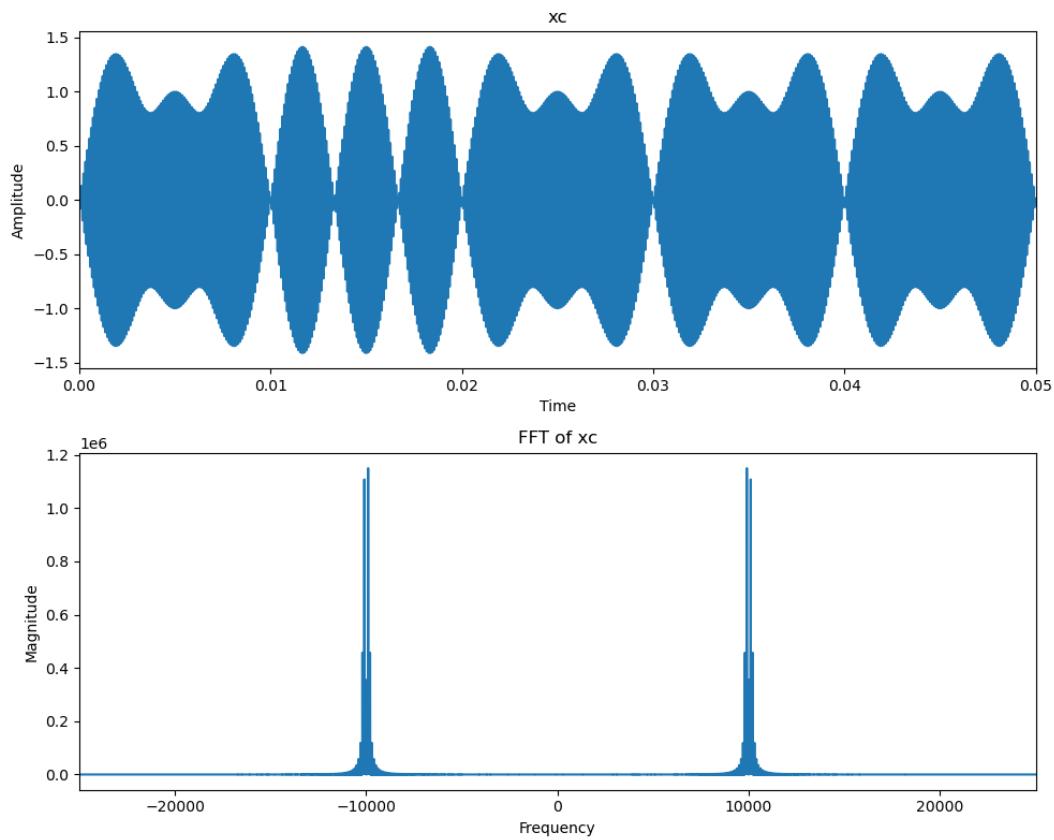
در فایل پروژه گفته شده است که با فرکانس ۱k و ۱.۵k انجام شود ولی به دلیل مرکز باند کانال داده شده و فرکانس حامل و پهنازی باند کانال داده شده میتوان محاسبه کرد و دید که این سیگنال در باند مورد نظر قرار نمیگیرد و حذف می شود و پیام را نمیتوان بازیابی کرد. (به این موضوع در گروه درس و پیوی تی ای مربوطه نیز اشاره شد) به همین علت این فرکانس ها ۱۵۰ و ۱۰۰ در نظر گرفته شدند. خروجی بلوک ها برای FSK :



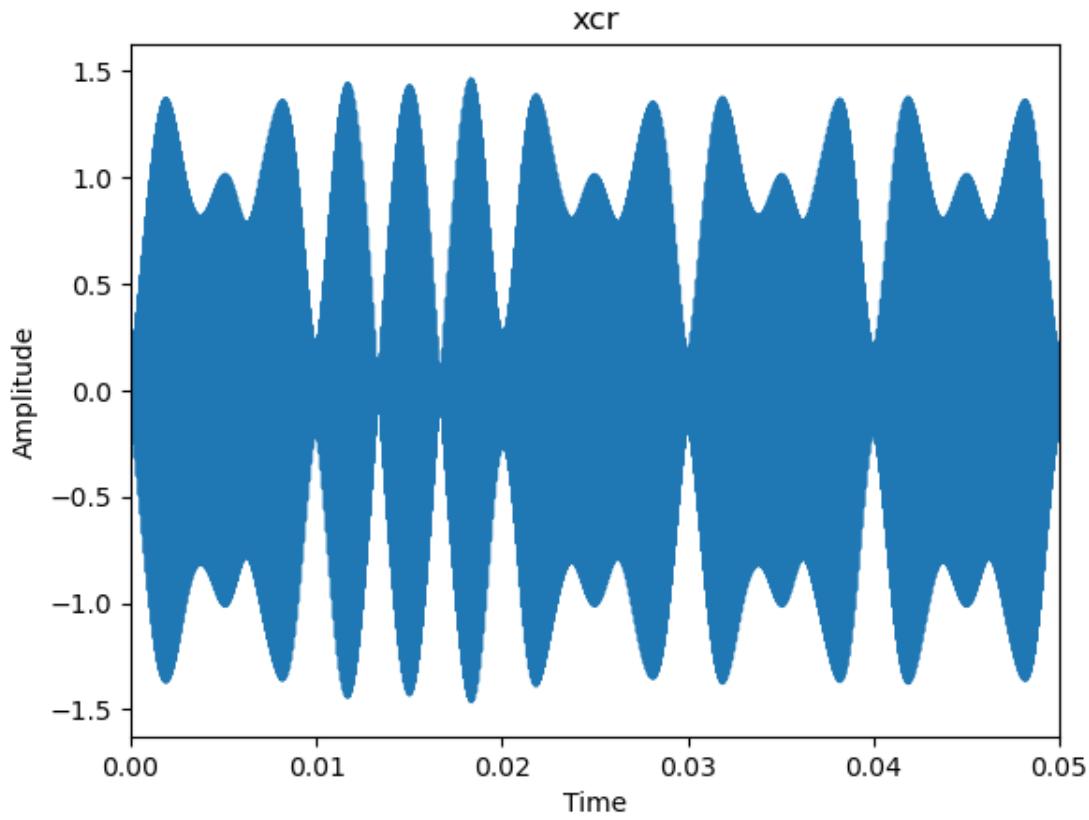
شکل ۳۰: خروجی بلوک Divider



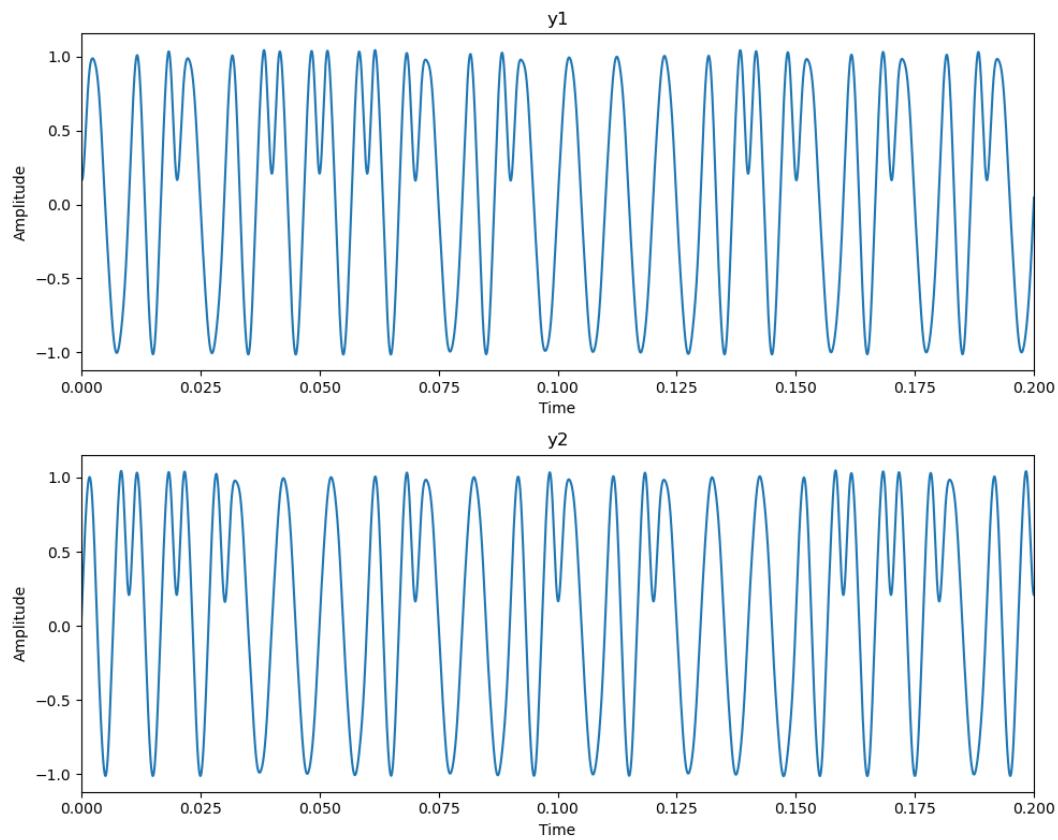
شکل ۳۱: خروجی بلوک Pulse Shaping



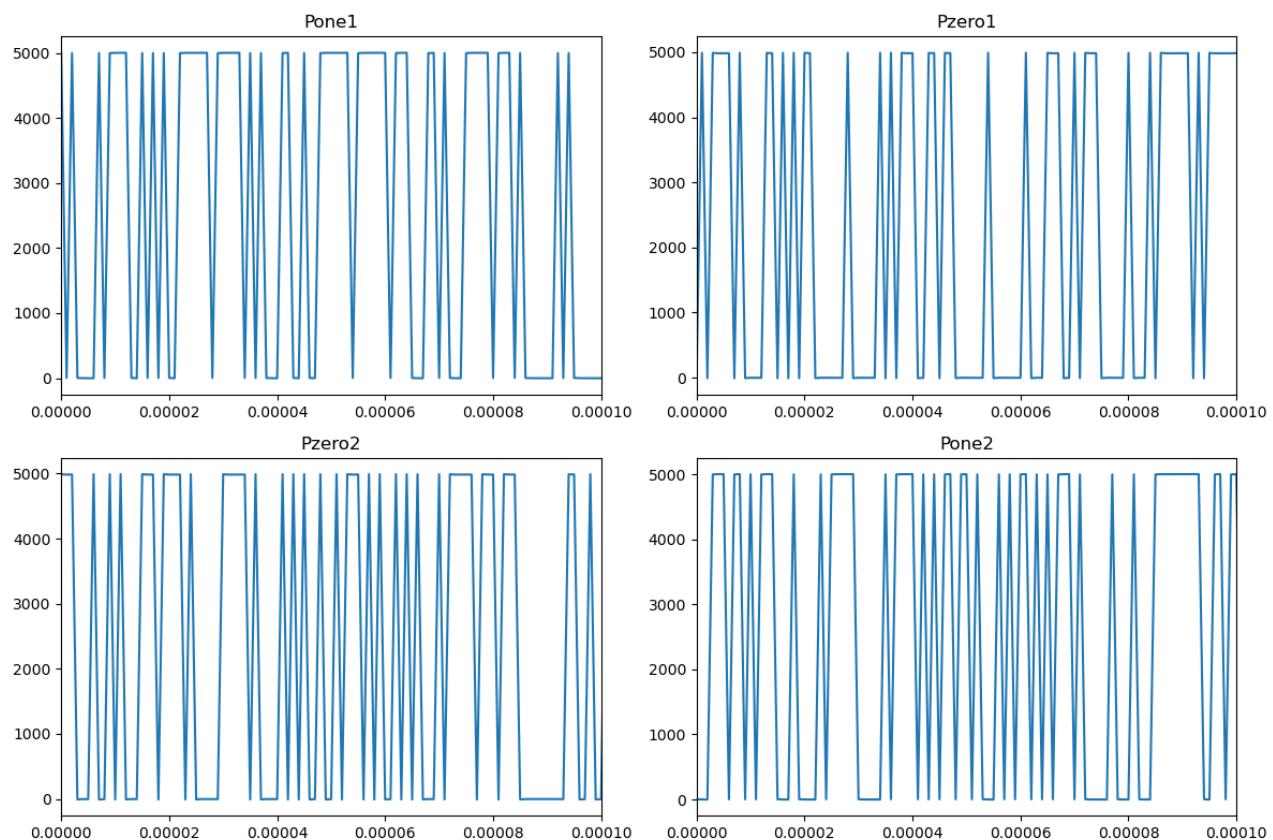
شکل ۳۲: خروجی بلوک AnalogMod



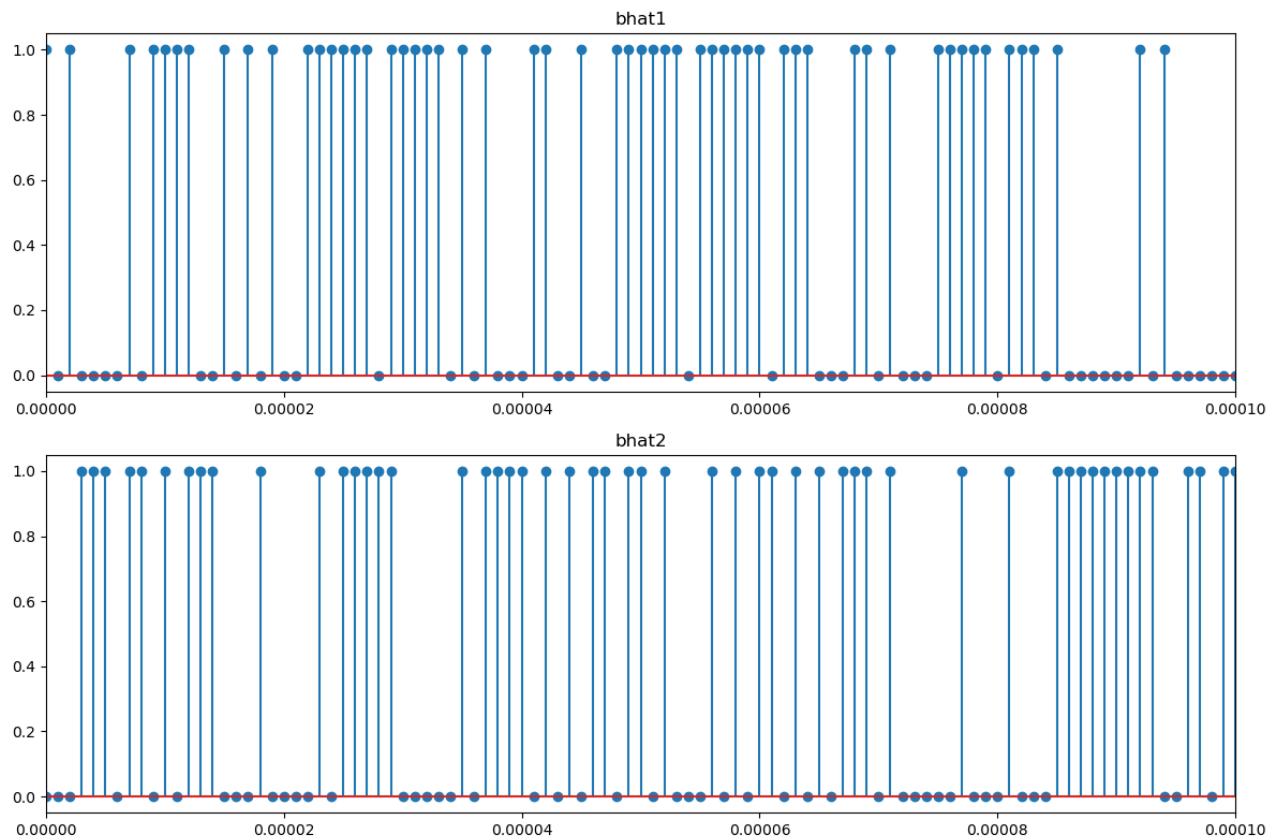
شکل ۳۳: خروجی بلوک Channel



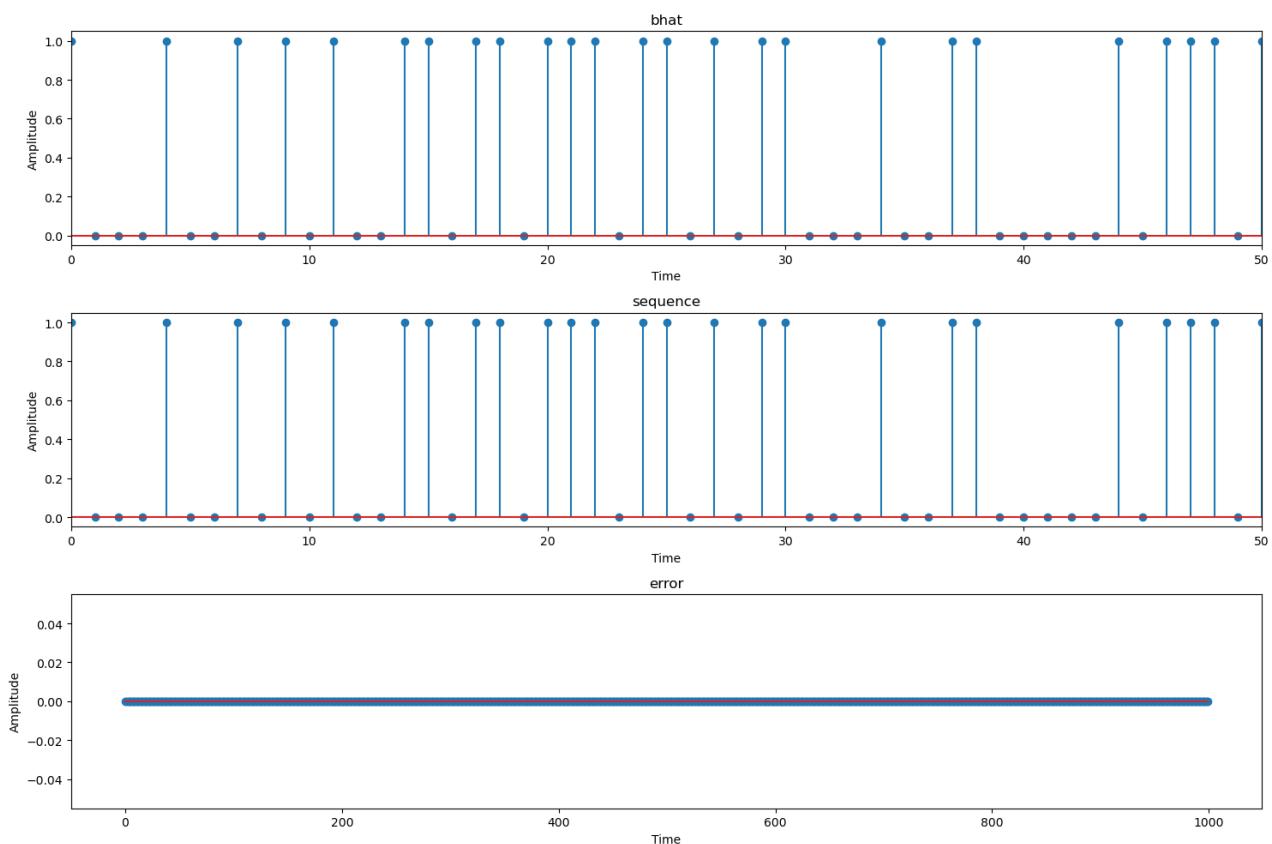
شکل ۳۴: خروجی بلوک AnalogDemod



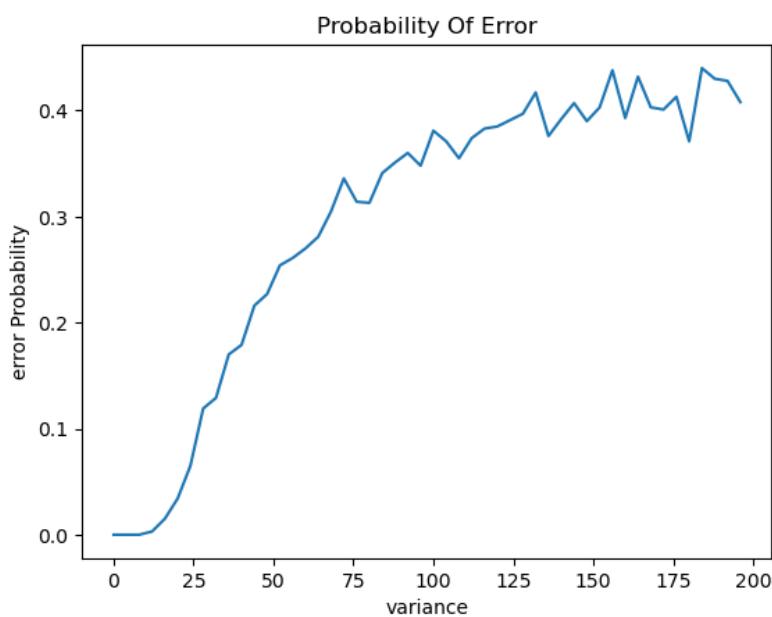
شکل ۳۵: خروجی بلوک Matched Filter



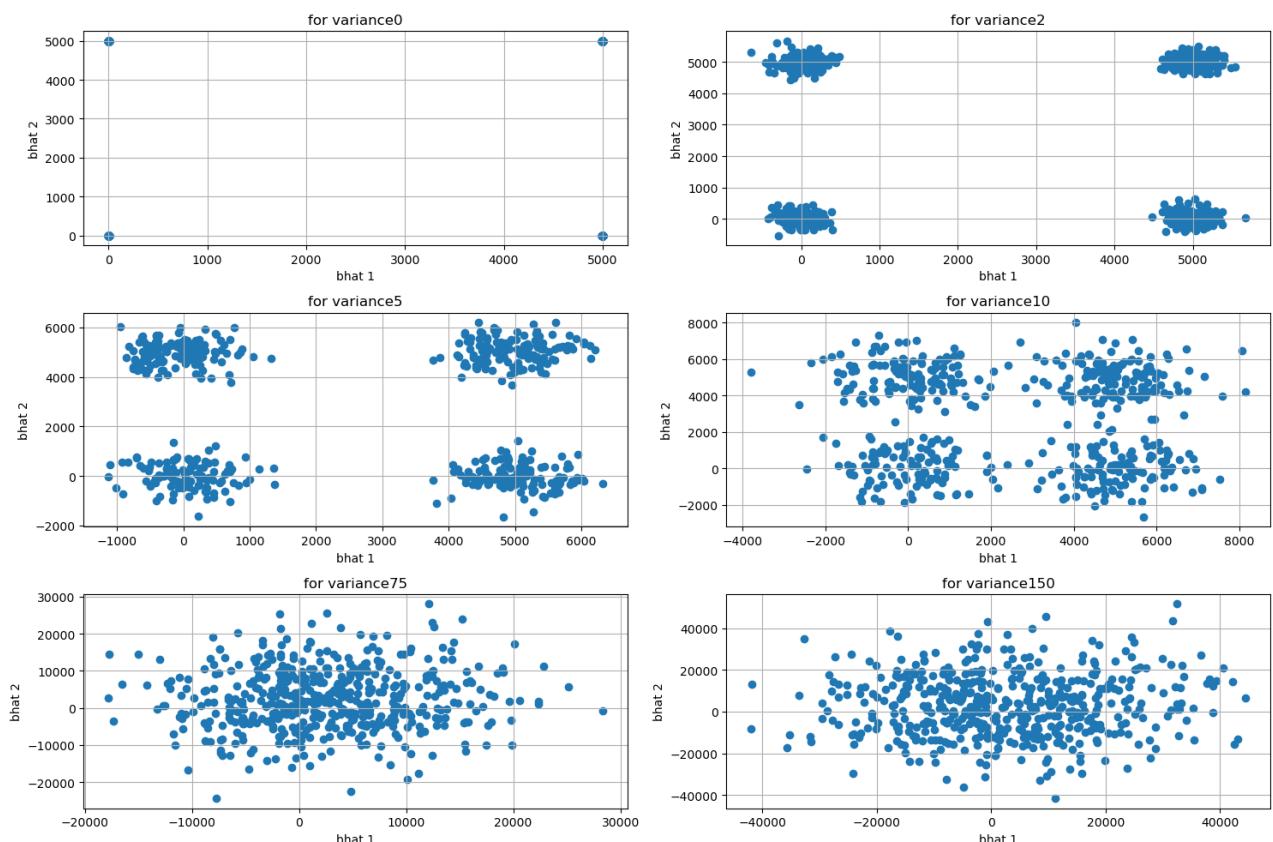
شکل ۳۶: خروجی بلوک Matched Filter



شکل ۳۷: خروجی بلوک Combine: سیگنال بازیابی شده



شکل ۳۸: نمودار احتمال خطأ بر حسب واریانس



شکل ۳۹: Signal Constellation

## ۴.۳

همانطور که در درس خواندیم و از خروجی نمودار های واریانس خطای نیر مشخص است کمترین خطای مربوط به سیستم PAM است. در نتیجه اگر معیار مقاومت نسبت به نویز است انتخاب ما PAM است.

اما هزینه استفاده از سیستم PAM مصرف توان بیشتر است که نسبت به دو روش دیگر توان بیشتری لازم دارد و کمترین توان مصرفی نیز مربوط به سیستم PSK است.

البته در آشکارسازی برای PSK تنها میتوانیم از آشکارساز coherent استفاده کنیم که نیاز به اسیلاتور دقیق و مچ شدن فرکانس دارد که هزینه بیشتری نیز میرد. ولی دو سیستم دیگر با روش آشکارسازی envelope detector که روش ساده تر و ارزان تری است نیز امکان آشکارسازی دارند.

همچنین اگر معیار مورد نظر ما پهنهای باند مصرفی باشد دو روش PAM و PSK بهتر از FSK هستند چون سیستم FSK پهنهای باند بیشتری مصرف میکند. اگر بخواهیم هر دو معیار توان و مقاومت نویز را با هم در نظر بگیریم سیستم PSK انتخاب معقول تری از بقیه است. البته در نهایت باید گفت که در هرجا بسته به آنکه معیار ما و هزینه هایی که حاضریم بگیریم و منابعی که در اختیار داریم سیستم مورد نظر برای انتخاب معقول تر ممکن است متفاوت باشد و سیستم ها به صورت کلی بهتر و بدتر نیستند ولی از جنبه های مختلف در بالا با یکدیگر بررسی شدند.

## ۴ انتقال دنباله ای از اعداد ۸ بیتی

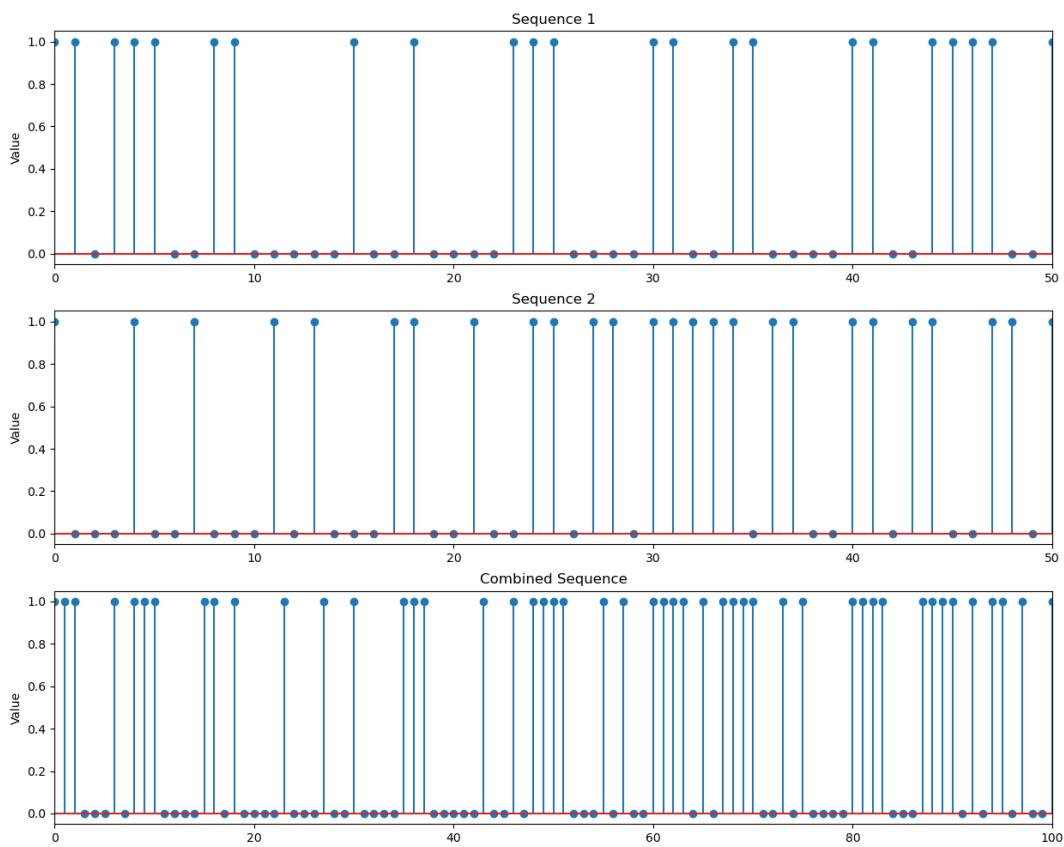
## SourceGenerator,OutputDecoder ۱.۴

```
def SourceGenerator(sequence):
    binary_sequence = [bin(num)[2:].zfill(8) for num in sequence]
    binary_string = ''.join(binary_sequence)
    binary_list = [int(bit) for bit in binary_string]
    return binary_list

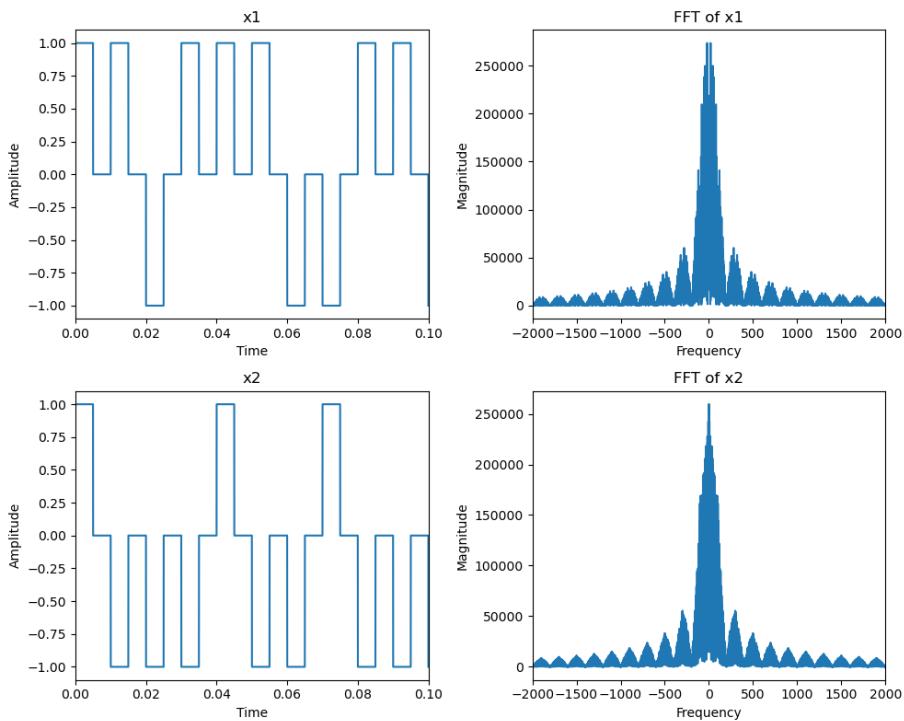
def OutputDecoder(binary_list):
    binary_string = ''.join(str(bit) for bit in binary_list)
    binary_sequence = [int(binary_string[i:i+8], 2) for i in range(0, len(binary_string), 8)]
    return binary_sequence
```

شکل ۴۰: SourceGenerator,OutputDecoder functions

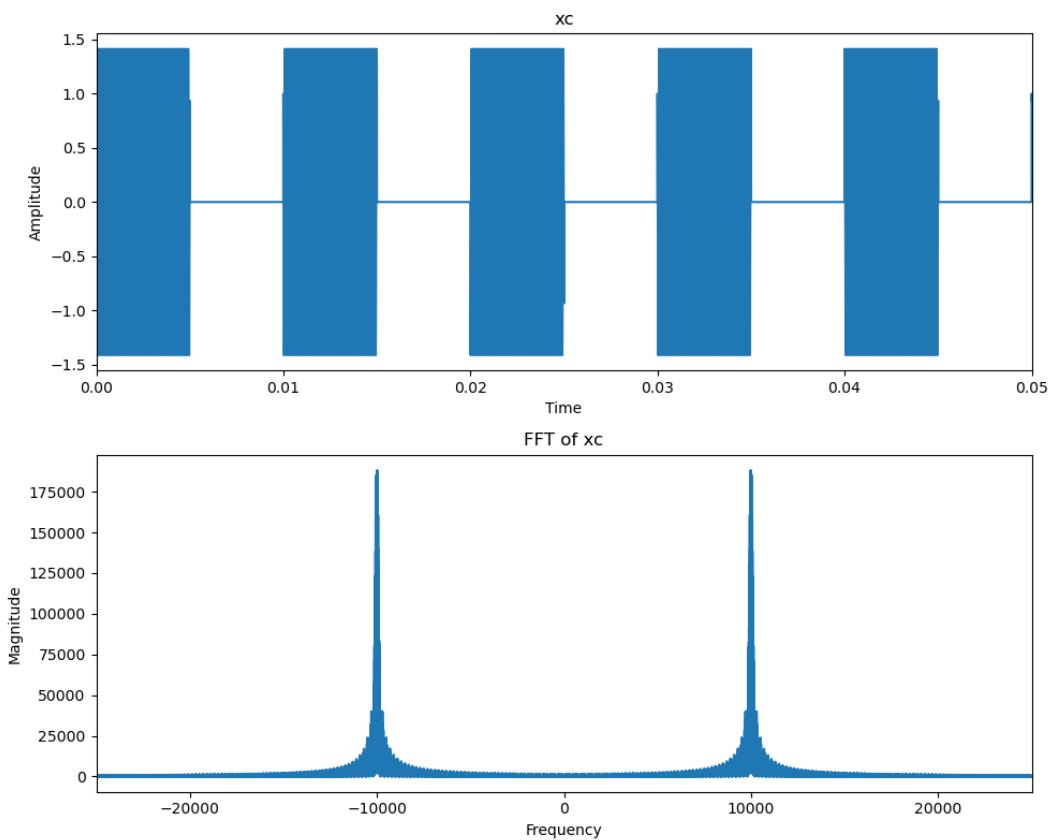
## Block output ۲.۴



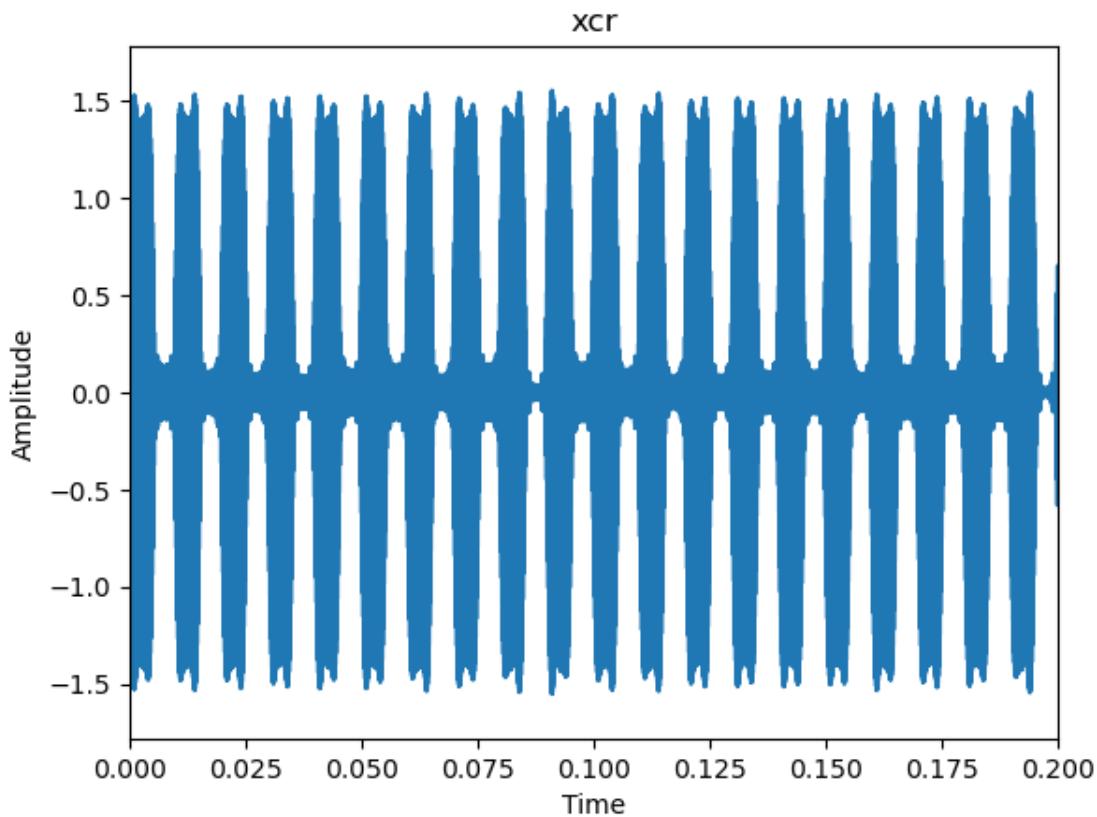
شکل ۴۱: خروجی بلوک Divider



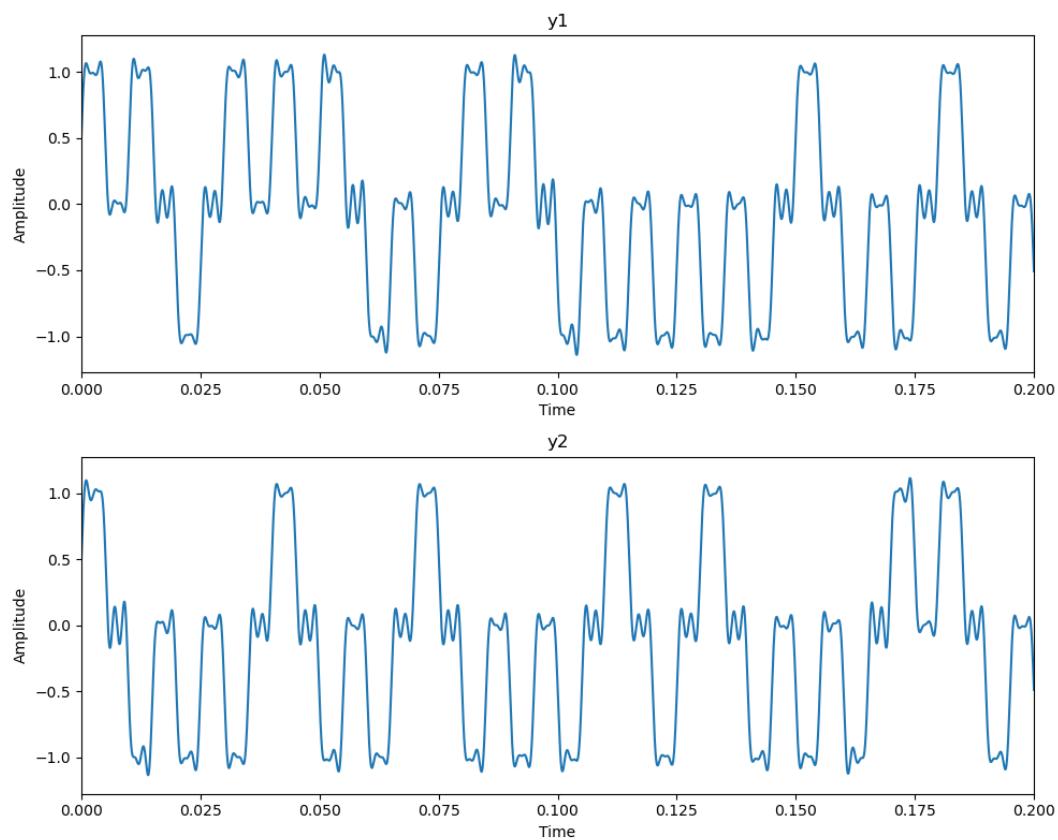
شکل ۴۲: خروجی بلوک Pulse Shaping



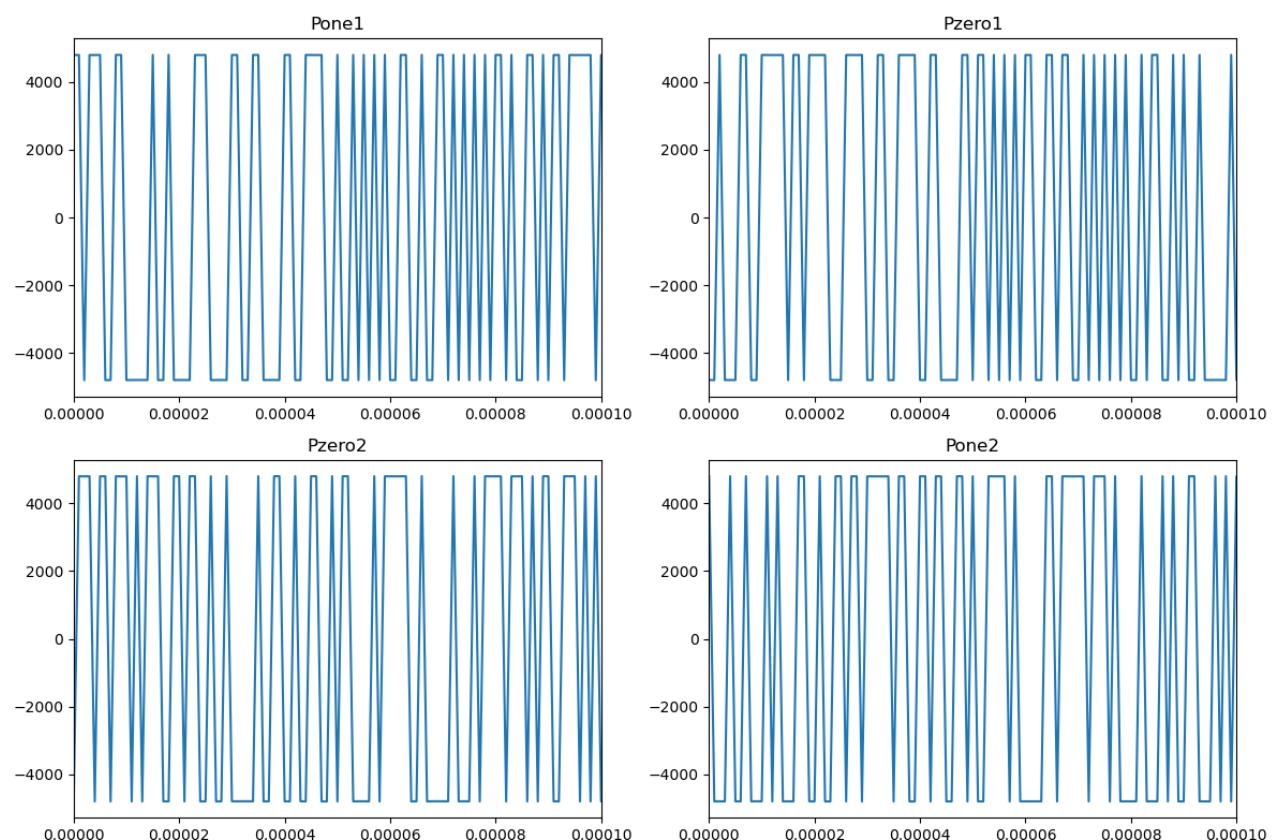
شکل ۴۳: خروجی بلوک AnalogMod



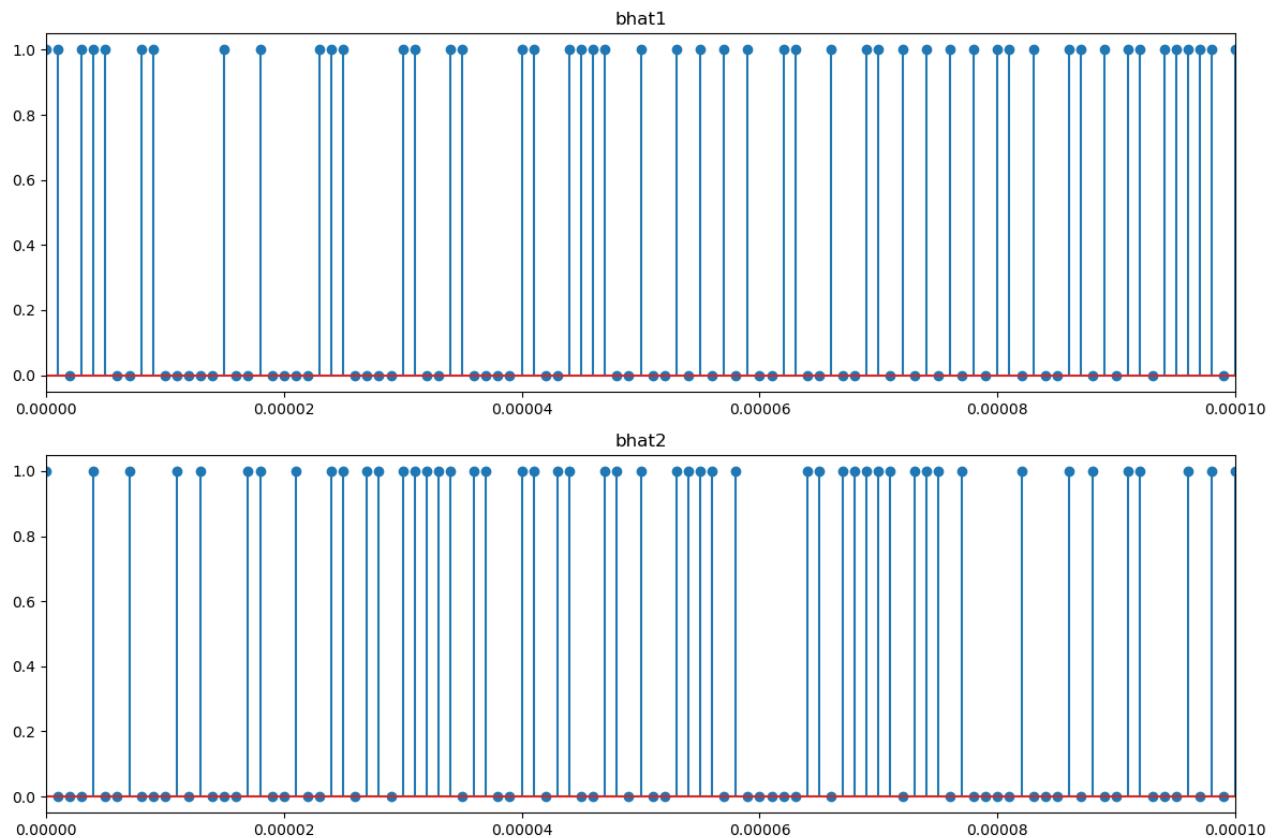
شکل ۴۴: خروجی بلوک Channel



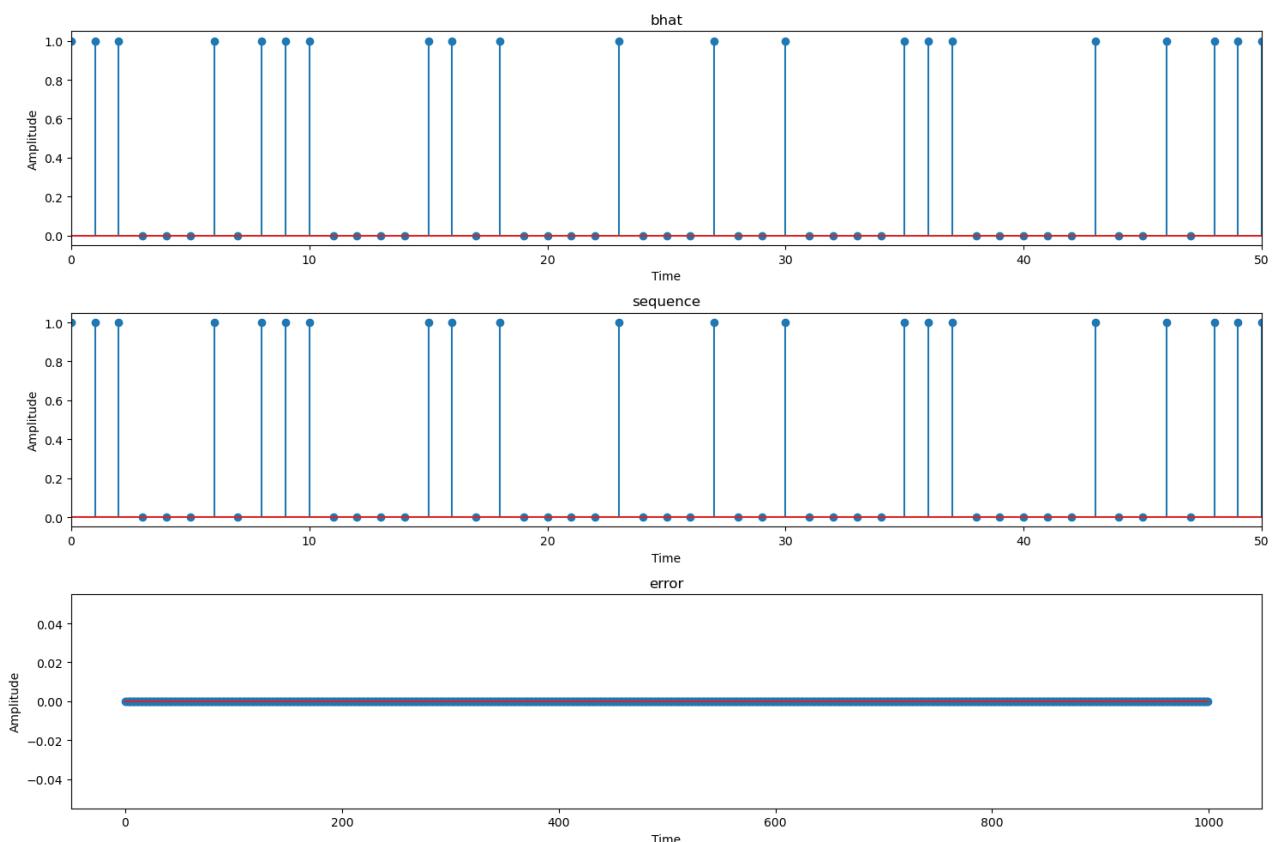
شکل ۴۵: خروجی بلوک AnalogDemod



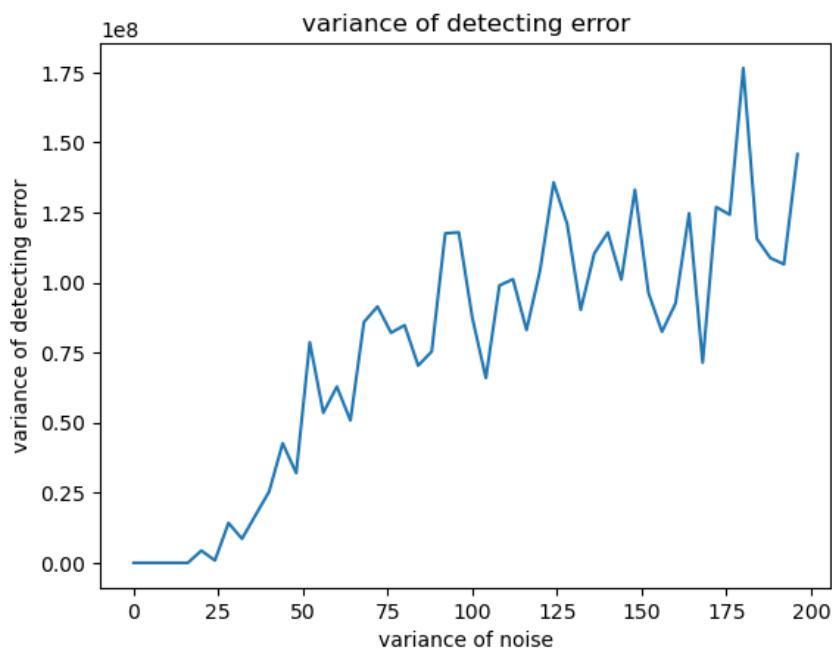
شکل ۴۶: خروجی بلوک Matched Filter



شکل ۴۷: خروجی بلوک Matched Filter



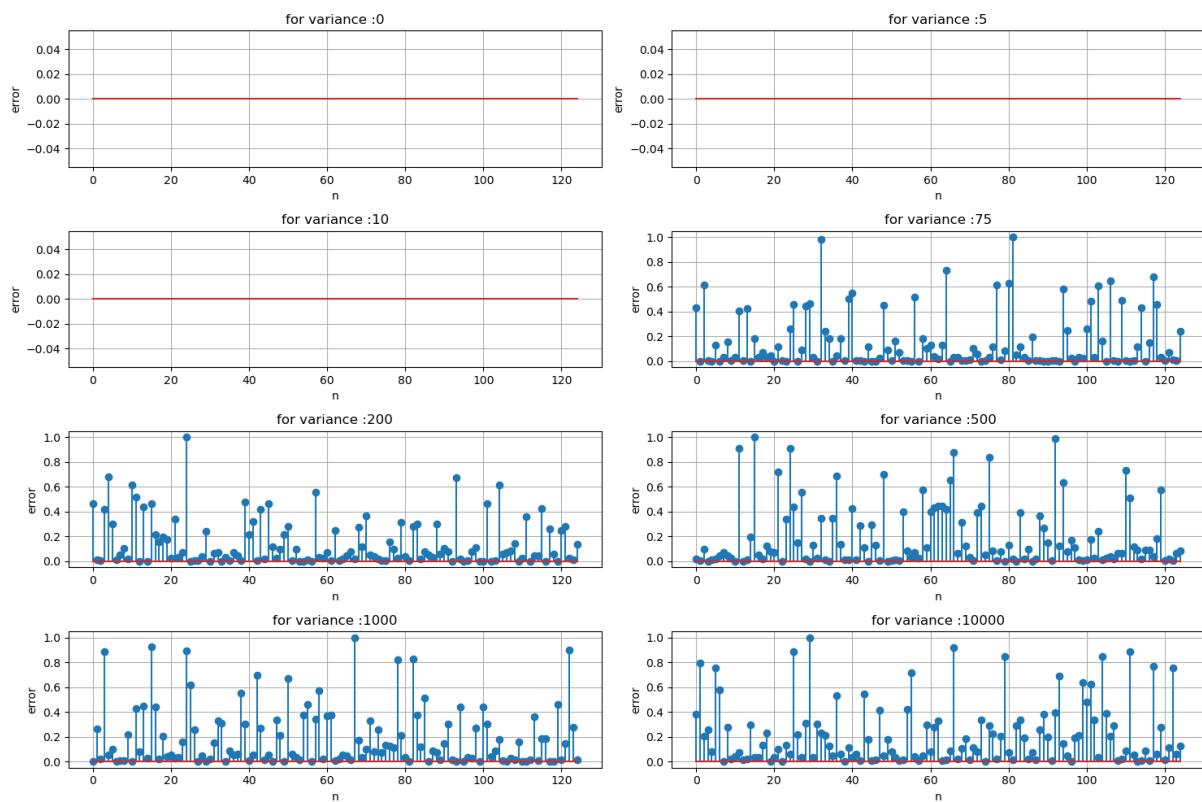
شکل ۴۸: خروجی بلوک Combine: سیگنال بازیابی شده



شکل ۴۹: نمودار واریانس خطای خروجی بر حسب واریانس نویز

شکل کلی نمودار با بخش های قبلی یکسان است یعنی با افزایش واریانس نویز، واریانس خطای تشخیص نیز افزایش می یابد. اما نمودار در این بخش یک تفاوت اسای دارد که مقادیر محور عمودی آن است که خیلی خیلی بزرگتر از بخش های قبلی شده است که دلیل این اتفاق این است که در بخش های قبلی فقط با ۰ و ۱ کار داشتیم ولی در این بخش چون اعداد کد شده بین ۰ تا ۲۵۵ هستند مقادیر واریانس خیلی بزرگتر شده است که همینگونه نیز انتظار میرفت.

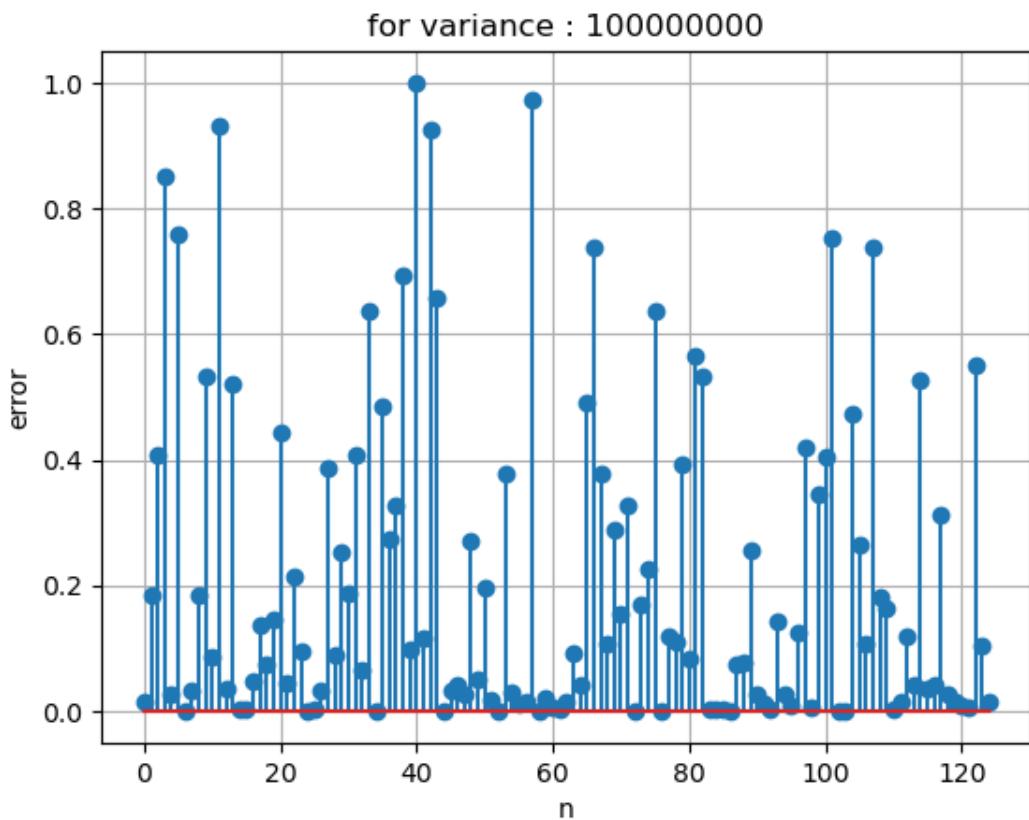
## Error distribution for different variances ۳.۴



شکل ۵۰: توزیع خطأ بر حسب مقادیر مختلف واریانس نویز

مشاهده میشود که برای واریانس های کم خیلی اروری کمی داریم و تقریبا در همه موارد به درستی بیت را تشخیص میدهیم ولی در واریانس های بزرگ تعداد خطای رخ داده و واریانس آنها خیلی زیاد میشود که این موضوع را میتوان از روی نمودار توزیع واریانس خطاهای مشاهده کرد.

## Error distribution for infinity variance ۴.۴



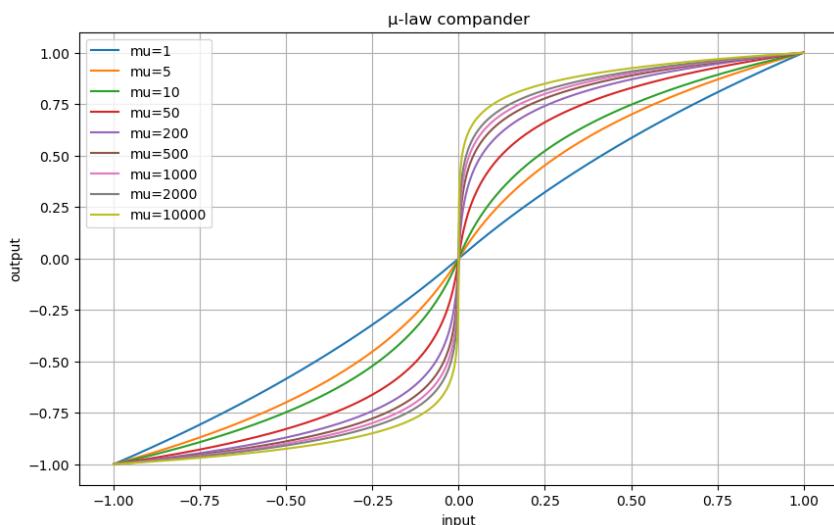
شکل ۵۱: توزیع خطا در حالت حدی میل واریانس به بی نهایت

به دلیل اینکه توزیع های ما به صورت یونیفرم هستند و توزیع نهایی حاصل ترکیب دو توزیع یونیفرم است میتوان گفت که توزیع نهایی بدست آمده به توزیع مثلثی میل میکند (تابعی که حاصل کانوالو شدن دو پالس مستطیلی است).

## ۵ فشرگستر

۱.۵

دامنه ورودی را بین  $-1$  و  $1$  فرض میکنیم و تابع ذکر شده را پیاده سازی میکنیم. سپس مقدار میو را مطابق آنچه که در لیل پلات مشخص است تغییر میدهیم تا تاثیر این پارامتر را مشاهده کنیم:



شکل ۵۲: رسم نمودار compander برای میو های مختلف

هر چه میو کمتر باشد این نمودار به حالت خطی نزدیک تر میشود. به عبارتی هر چه مقدار میو کمتر باشد مقادیر به صورت خطی مپ میشوند ولی اگر میو را بزرگ کنیم فاصله بین استپ ها مقادیر بزرگ بیشتر میشوند. یعنی پله های مپ کردن در اعداد پایین به یکدیگر نزدیکتر هستند ولی در اعداد بزرگتر این فاصله بیشتر میشود و پینگک به صورت غیرخطی انجام میشود.

## ۲.۵

ابتدا تابعی مینویسیم تا با استفاده از آن بتوانیم صدای خود را ضبط کنیم و فایل آن را ذخیره کنیم (مشابه کاری که در تمرین کامپیوتری ۲ انجام شد). تابع پیاده شده به صورت زیر است :

```
# Function to record audio
def audio_recorder(time, sample_rate):
    """Record audio for a specific time and sample rate."""
    recorded_audio = sounddevice.rec(int(time * sample_rate), samplerate=sample_rate, channels=1)
    sounddevice.wait() # Wait for the recording to end
    return recorded_audio

# Record audio for 20 seconds
audio = audio_recorder(60, 48000)
# Save the recorded audio as a .wav file
wavfile.write('audio', 48000, audio)
```

شکل ۵۳: تابع ضبط و ذخیره صوت

لازم به ذکر است که مدت زمان صوت ذخیره شده ۱ دقیقه است. اگر میخواهید کد ها را ران کنید حواستان باشد که این بخش نباید ران شود چون فایل صوتی ذخیره شده دوباره ضبط میشود و نتایج از دسترس خارج میشود. برای نمونه برداری از صوت طبق اسلاید های درس میتوان از فرکانس ۴۸۰۰۰ یا ۴۴۱۰۰ استفاده کرد. تابع اجرای صوت ذخیره شده و نمایش دادن آن به صورت زیر است :

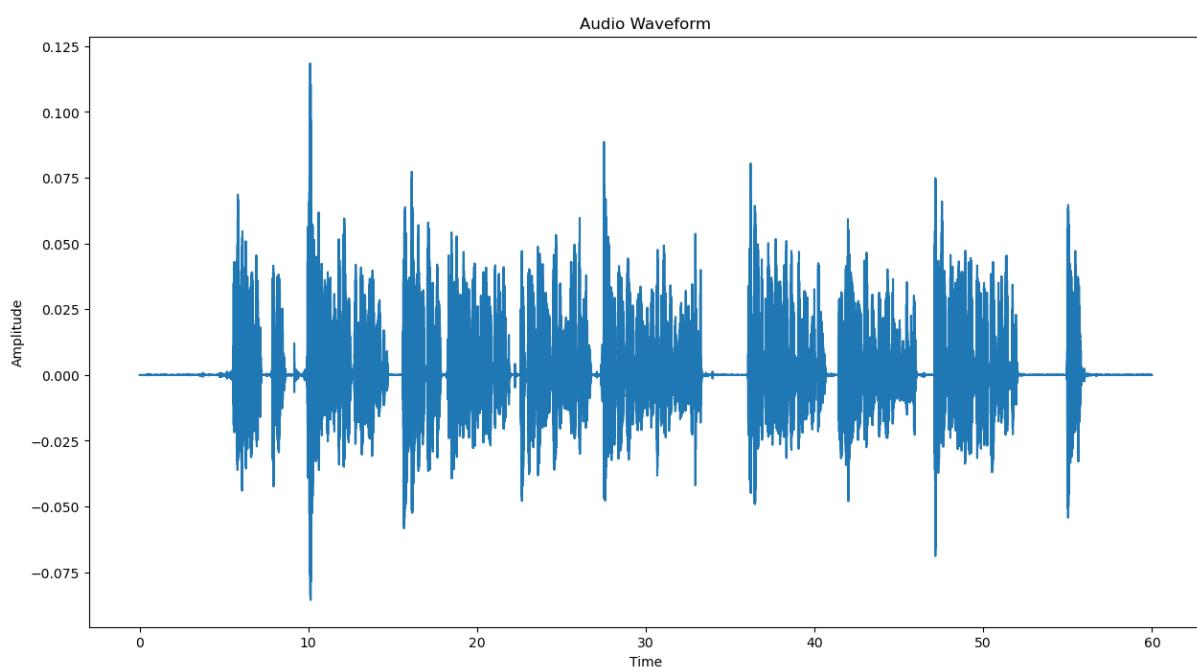
```

# Load the audio file
sample_rate, audio = wavfile.read('audio.wav')
audio_t = np.arange(0, len(audio))/sample_rate
# Play the audio
sounddevice.play(audio, sample_rate)

# Plot the audio waveform
plt.plot(audio_t, audio)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Audio Waveform')
plt.show()

```

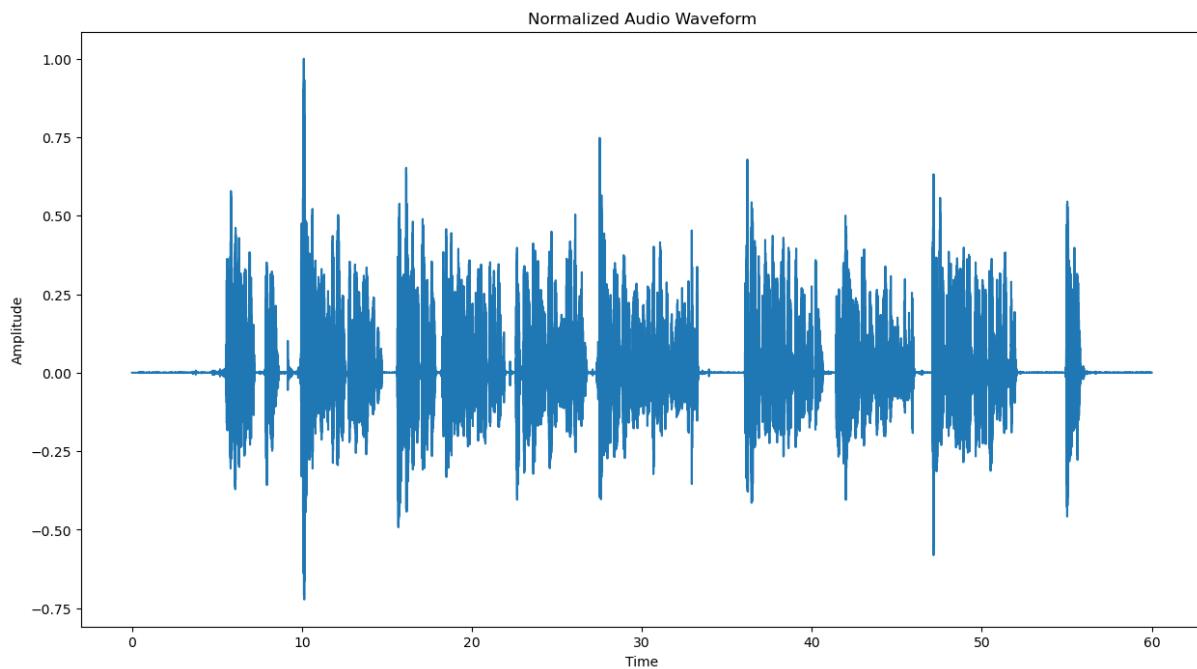
شکل ۵۴: تابع پخش صوت و نمایش آن



شکل ۵۵: نمایش صوت ذخیره شده

## ۳.۵

برای نرمالیزه کردن صوت مقادیر آن را بر ماکسیمم مقدار آن تقسیم میکنیم تا بزرگترین دیتا ما به ۱ مپ شود (یا کوچکترین دیتا به -۱ مپ شود)



شکل ۵۶: صوت نرمالایز شده

محاسبه توان سیگنال نرمالایز شده:

calculating audio power

```

power = np.mean(np.abs(normalized_audio) ** 2)
power_db = 10 * np.log10(power)
print("Normalized Audio Power (dB):", power_db)

✓ 0.0s

Normalized Audio Power (dB): -24.682424068450928

```

شکل ۵۷: توان صوت نرمالایز شده

۴.۵

پیاده سازی تابع خواسته شده به سادگی با استفاده از رابطه داده شده در اول این بخش انجام میشود:

```
def ulaw_compressor(signal, mu):
    companded_signal = []
    for s in signal:
        companded_signal.append(np.sign(s) * np.log(1 + mu * np.abs(s)) / np.log(1 + mu))
    return np.array(companded_signal)
```

شکل ۵۸: تابع compressor

۵.۵

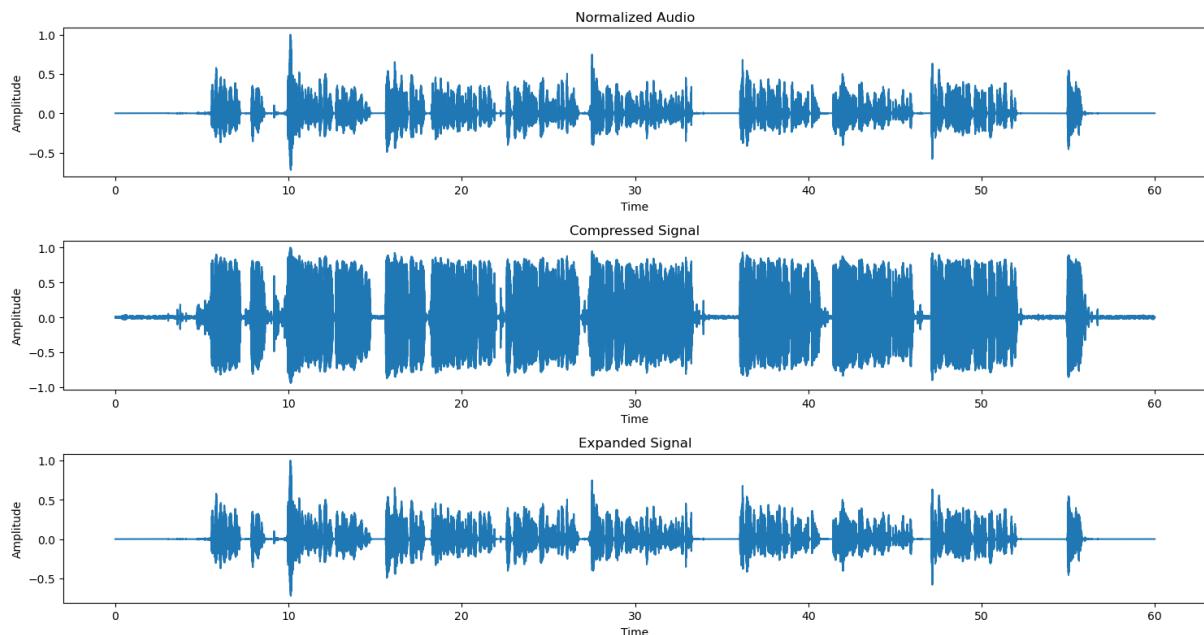
رابطه وارون تابع قبلی را با استفاده از اینترنت و اسلاید های درس به دست آوردهیم و تابع معکوس را پیاده سازی کردیم :

```
def ulaw_expander(companded_signal, mu):
    expanded_signal = []
    for s in companded_signal:
        expanded_signal.append(np.sign(s) * (1 / mu) * ((1 + mu) ** np.abs(s) - 1))
    return np.array(expanded_signal)
```

شکل ۵۹: تابع expander

۶.۵

در این بخش سیگنال صوت نرمالایز شده را ابتدا compress و سپس expand میکنیم. سیگنال ابتدایی و پس از عبور از هر بلوک در زیر نمایش داده شده است :



شکل ۶۰: تابع expander

همانطور که قابل مشاهده است سیگنال بازیابی شده تطابق خوبی با سیگنال اولیه دارد. خطای RMS برای مقادیر میو های مختلف :

```
RMS reconstruction Error for mu = 1e-06 : 6.221533586197459e-11
RMS reconstruction Error for mu = 1 : 7.401486830834377e-19
RMS reconstruction Error for mu = 5 : 6.230381449772144e-18
RMS reconstruction Error for mu = 50 : 1.025456452771753e-17
RMS reconstruction Error for mu = 255 : 1.7148512870603542e-17
RMS reconstruction Error for mu = 2000 : 2.061682401891981e-17
RMS reconstruction Error for mu = 10000 : 4.548278365211866e-17
```

شکل ۶۱: تابع expander

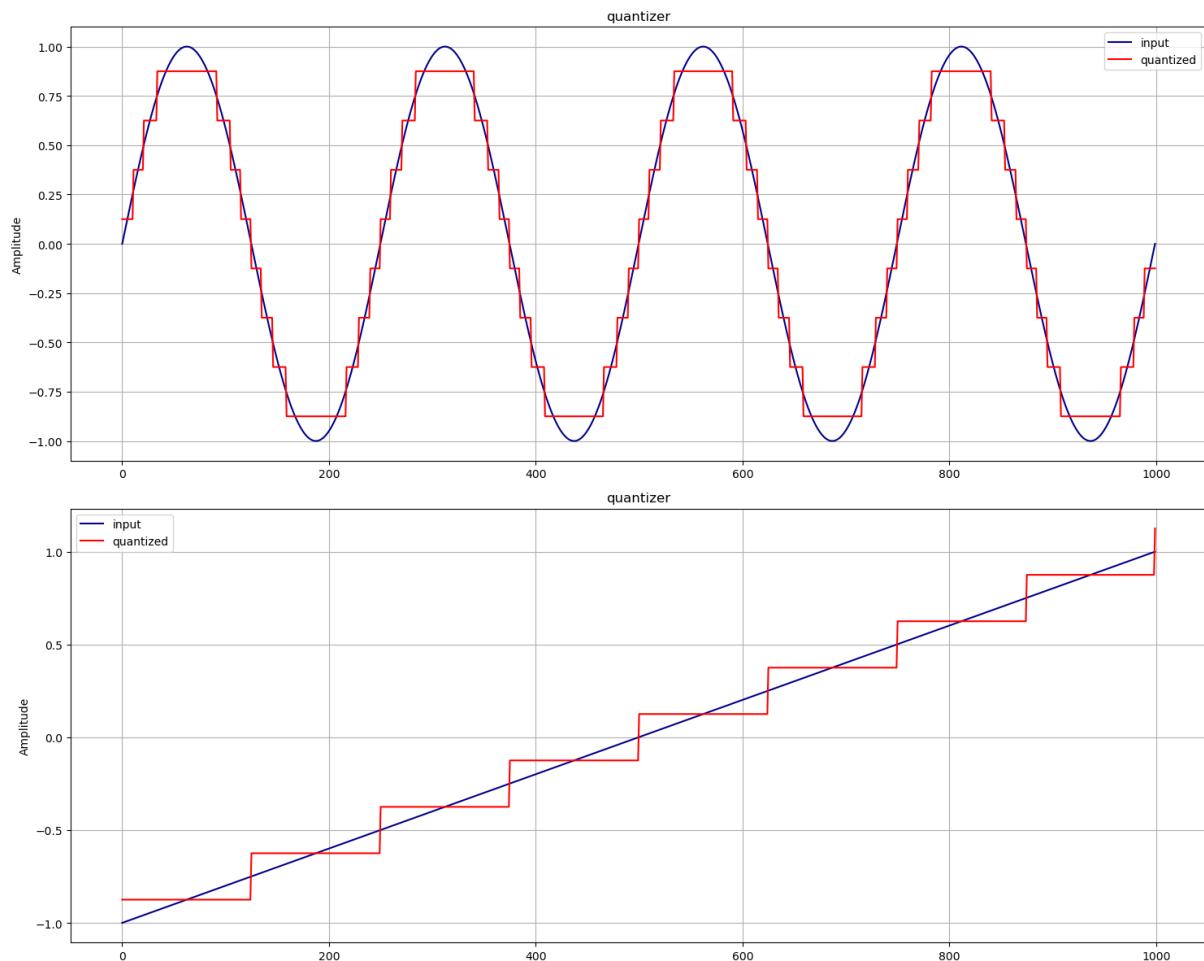
## ۷.۵

تابع پیاده شده quantizer :

```
def quantizer(signal, num_levels):
    # Calculate the step size for quantization
    step_size = 2 / num_levels
    # Quantize the signal
    quantized_signal = (np.floor(signal / step_size) + 0.5) * step_size
    return quantized_signal
```

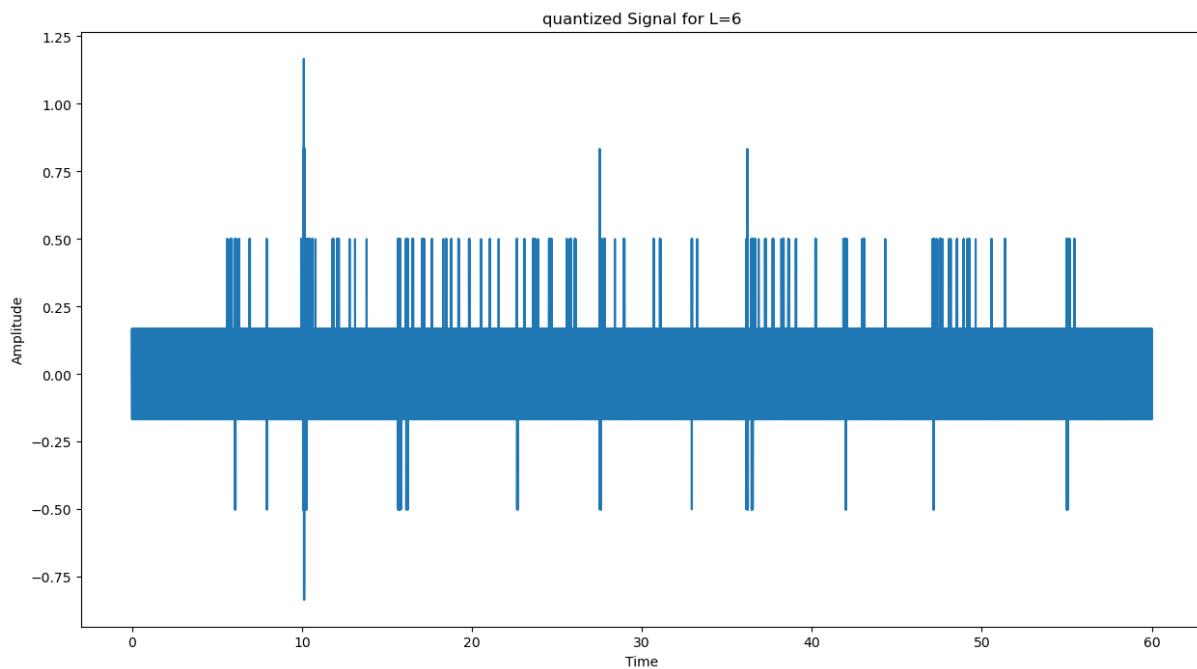
شکل ۶۲: تابع quantizer

خروجی تابع کوانتايزر برای دو مثال خطی و سینوسی :



شکل ۶۳: اثر تابع بر روی دو مثال

اثر عملکرد تابع روی سیگنال صوت نرمالایز شده برای  $L=6$  :



شکل ۶۴: اثر تابع کوانتايزر بر روی صوت ورودی نرمالایز شده

#### ۸.۵

تابع محاسبه SNR

```
def calculate_snr(input_signal, output_signal):
    # Calculate the power of the input signal
    input_power = np.mean(np.abs(input_signal) ** 2)

    # Calculate the power of the quantization error (difference between input and output signals)
    quantization_error = input_signal - output_signal
    quantization_error_power = np.mean(np.abs(quantization_error) ** 2)

    # Calculate the SNR in dB
    snr_db = 10 * np.log10(input_power / quantization_error_power)

    return snr_db
```

شکل ۶۵: تابع محاسبه SNR

حال صوت نرمالایز شده را از این دو بلوک به ازای مقادیر مختلف کوانتايزشن عبور میدهیم و SNR سیگنال خروجی را برای این مقادیر مختلف نشان میدهیم :

```
SNR for quantization level 4: -11.792457103729248 dB
SNR for quantization level 5: -9.701957106590271 dB
SNR for quantization level 6: -7.990593910217285 dB
SNR for quantization level 7: -6.546003818511963 dB
SNR for quantization level 8: -5.298458933830261 dB
```

شکل ۶۶: SNR's

همانطور که از آنچه که در درس آموختیم نیز انتظار میرفت با افزایش سطوح کوانتیزیشن مقدار SNR سیگنال خروجی افزایش می یابد. یعنی ما هر چه از تعداد سطوح بیشتری استفاده کنیم سیستم ساخته شده مقاومت نسبت به نویز بهتری دارد.

### ۹.۵

در این بخش سیستم هم شامل دوتابع compressor و expander و هم شامل quantizer است. صوت نزمالایز شده را به این سیستم میدهیم و به ازای مقادیر میو و L مختلف SNR سیگنال خروجی را محاسبه میکنیم:

```

SNR for quantization level 4 and μ value 1e-06: -11.79245413260728 dB
SNR for quantization level 4 and μ value 1: -9.184991182309803 dB
SNR for quantization level 4 and μ value 5: -4.482252967275084 dB
SNR for quantization level 4 and μ value 50: 0.39156631770368716 dB
SNR for quantization level 4 and μ value 255: -0.19806143581618116 dB
SNR for quantization level 4 and μ value 2000: 0.7346577306423098 dB
SNR for quantization level 4 and μ value 10000: 1.9121751375316365 dB
SNR for quantization level 5 and μ value 1e-06: -9.701950756430133 dB
SNR for quantization level 5 and μ value 1: -6.931559859205602 dB
SNR for quantization level 5 and μ value 5: -2.0937894429723096 dB
SNR for quantization level 5 and μ value 50: 4.106294933995341 dB
SNR for quantization level 5 and μ value 255: 2.9991481228996624 dB
SNR for quantization level 5 and μ value 2000: -3.3398755215567943 dB
SNR for quantization level 5 and μ value 10000: -7.992756434048786 dB
SNR for quantization level 6 and μ value 1e-06: -7.99058915886115 dB
SNR for quantization level 6 and μ value 1: -5.126819585668754 dB
SNR for quantization level 6 and μ value 5: -0.20740092692879852 dB
SNR for quantization level 6 and μ value 50: 5.71431898580527 dB
SNR for quantization level 6 and μ value 255: 3.0355531921533747 dB
SNR for quantization level 6 and μ value 2000: 0.45984112328457794 dB
SNR for quantization level 6 and μ value 10000: 0.08412249513832631 dB
SNR for quantization level 7 and μ value 1e-06: -6.54600063623743 dB
SNR for quantization level 7 and μ value 1: -3.6261809136245797 dB
SNR for quantization level 7 and μ value 5: 1.358195502821347 dB
SNR for quantization level 7 and μ value 50: 7.058425469656604 dB
...
SNR for quantization level 8 and μ value 50: 8.537108224493082 dB
SNR for quantization level 8 and μ value 255: 6.918576426004459 dB
SNR for quantization level 8 and μ value 2000: 2.9687844564997397 dB
SNR for quantization level 8 and μ value 10000: 1.05648547979415 dB

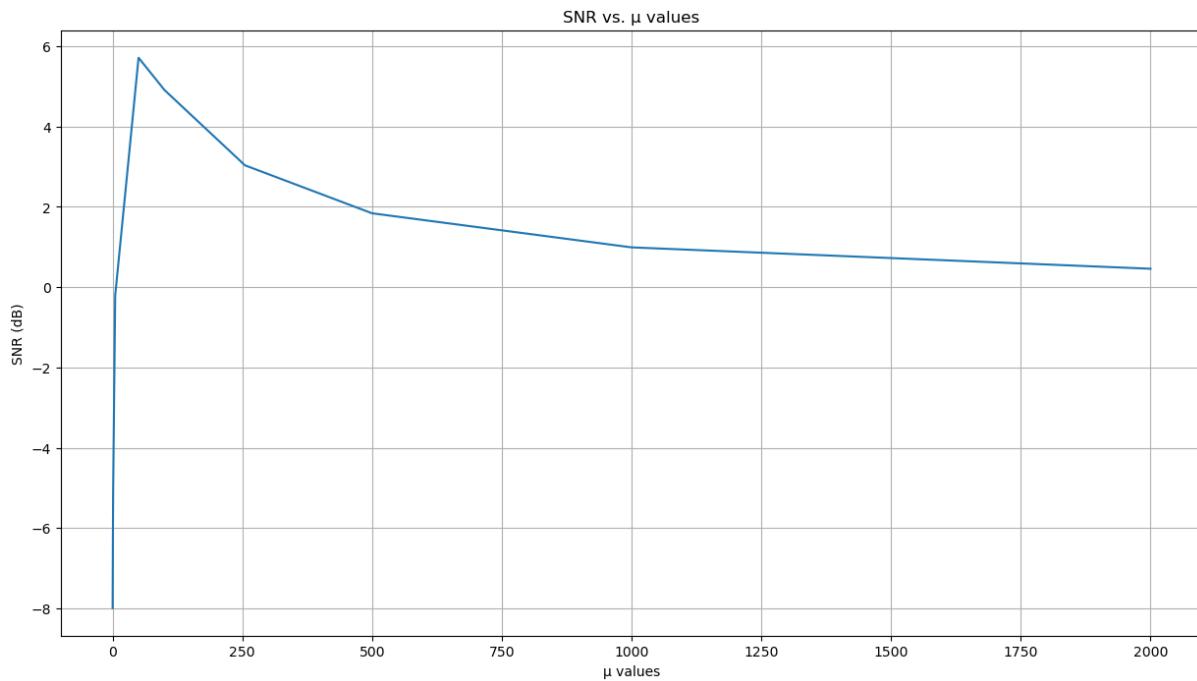
```

شکل ۶۷: SNR's

طبق بخش قبلی متوجه شدیم که با افزایش سطوح کوانتیزه کردن مقاومت به نویز بهتر است و SNR سیگنال خروجی بیشتر میشود. و همچنین مشاهده میشود که با تغییر مقدار میو SNR ابتدا افزایش و سپس کاهش می یابد. که این با منطق مانیز سازگار است که تا یک مقداری افزایش میو سیستم ما را بهبود می بخشد ولی اگر این مقدار را خیلی زیاد کنیم کلا سیگنال را در رنج آن نمی افتد و در اکثر مقداری دچار خطأ میشود و کار کرد سیستم رو به تضعیف میرود.

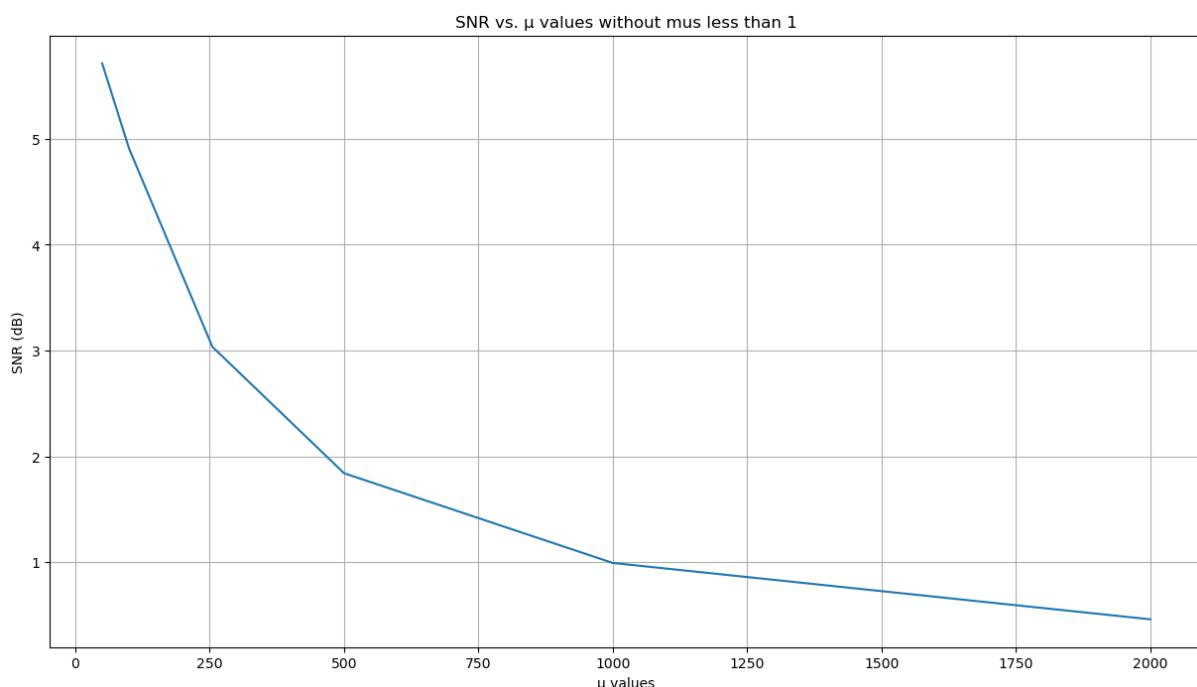
## ۱۰.۵

نمودار SNR بر حسب مقادیر میو :



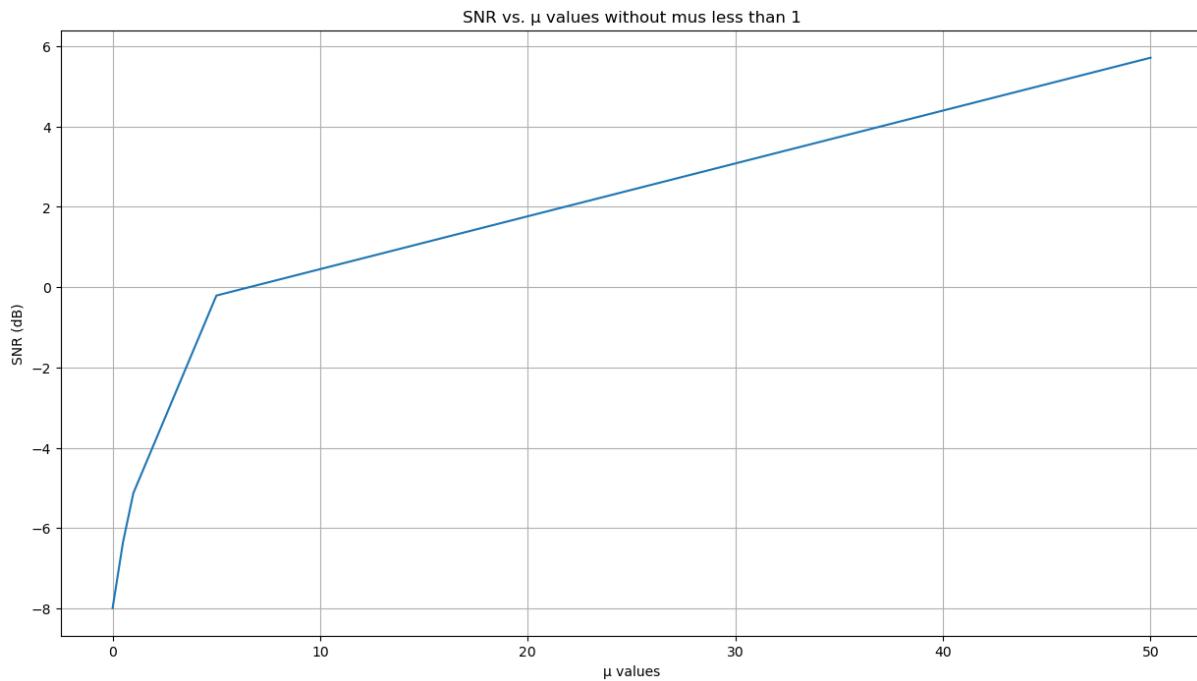
شکل ۶۸: SNR for different mu's

در نظر گرفتن فقط میو های بزرگتر از ۱ :



شکل ۶۹: SNR for different mu's

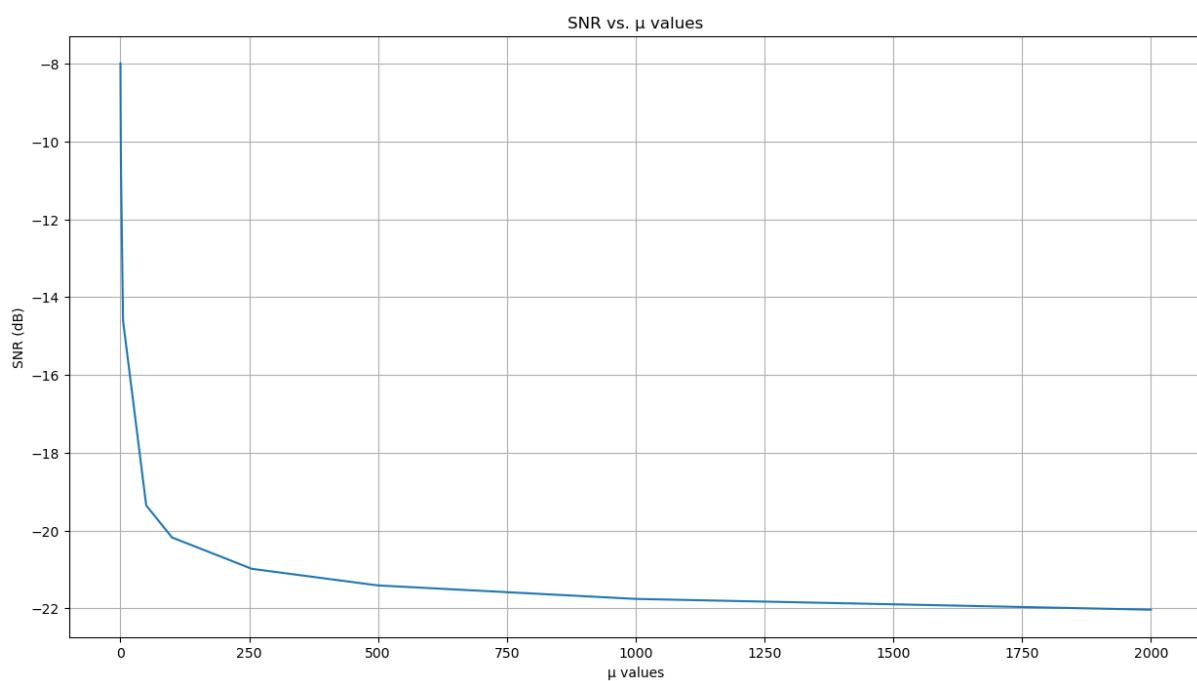
در نظر گرفتن فقط میو تا حد اکثر ۵۰ :



شکل ۷۰ SNR for different mu's

۱۱.۵

در این ترکیب برای میو های بزرگتر از ۱ استفاده از فشر گستر باعث کاهش SNR میشود ولی برای میو های کوچکتر از ۱ این سیستم باعث افزایش SNR میشود. بله این ترکیب نیز میتواند اثر مثبت در یک سری از مقادیر میو برای سیستم داشته باشد.



شکل ۷۱ SNR for different mu's

```
snrs
✓ 0.0s
[ -7.990596967290009,
-9.578253495139029,
-10.69599802587264,
-14.605695261304097,
-19.354714774451335,
-20.179449619446288,
-20.986034641326974,
-21.41290394892293,
-21.759326906256383,
-22.03676974804111]
```

شکل ۷۲: SNR's