به نام خدا

گزارش پروژه نهایی
سیستم‌های مخابراتی

مبین خطیب
۹۹۱۰۶۱۱۴

۱ مقدمه

در این قسمت دیاگرام فرستنده و گیرنده ترسیم شده که برای حل مسئله از آن استفاده کردیم

۲ پیاده سازی بلوک ها به صورت مجزا

۱.۲: دو تابع Divide و Combine را مطابق خواسته مساله پیاده سازی کردیم. تنها نکته مهم این است که برای نزدیک شدن به حالت Real-Time، درایه های فرد b را در ۱b و درایه های زوج را در ۲b قرار دادیم. اینگونه میتوان فرض کرد که دو پردازش همزمان و موازی هستند و سرعت ما دوبرابر شده و از طرفی همزمانی نیز رعایت میشود. در تابع Combine نیز همین منطق را در پیش گرفتیم.

۲.۲: تابع PulseShaping را مطابق خواسته سوال طراحی کردیم. به این صورت که به ازای هر بیت ۱ در آرایه باینری، یک شکل موج مختص به بیت برابر با ۱ و به ازای هر بیت ۰ در آرایه باینری، یک شکل موج مختص به بیت برابر با ۰ قرار دادیم. ترتیب این ها را نیز رعایت کردیم.

۳.۲: تابع AnalogMod را مطابق خواسته سوال طراحی کردیم. به این صورت که سیگنال ورودی را در توابع سینوس و کسینوس با فرکانس مرکزی f_c ضرب کردیم و پس از جمع با یکدیگر آن را خروجی دادیم. توابع سینوسی را با فرکانس f_s نمونه برداری کردیم.

۴.۲: فیلتر هارا به صورت دستی تولید و استفاده کردیم میدانیم که فیلتر حوزه زمان $sinc(t)$ فیلتر پایین گذر در حوزه زمان است. از طرفی طبق قوانین فوریه ای میدانیم $g(t) \leftrightarrow G(f) \Leftrightarrow g(t-t_0) \Leftrightarrow e^{-j2\pi ft_0}G(f)$ بنابر این با ضرب دو نمایی در سیگنال $sinc(t)$میتوانیم فیلتر میانگذر دلخواه را تولید کنیم. با عبور سیگنال $x_{transmit}$ از فیلتر ساخته شده سیگنال دریافت شده در دریافت کننده یعنی $x_{receive}$ را تولید میکنیم.

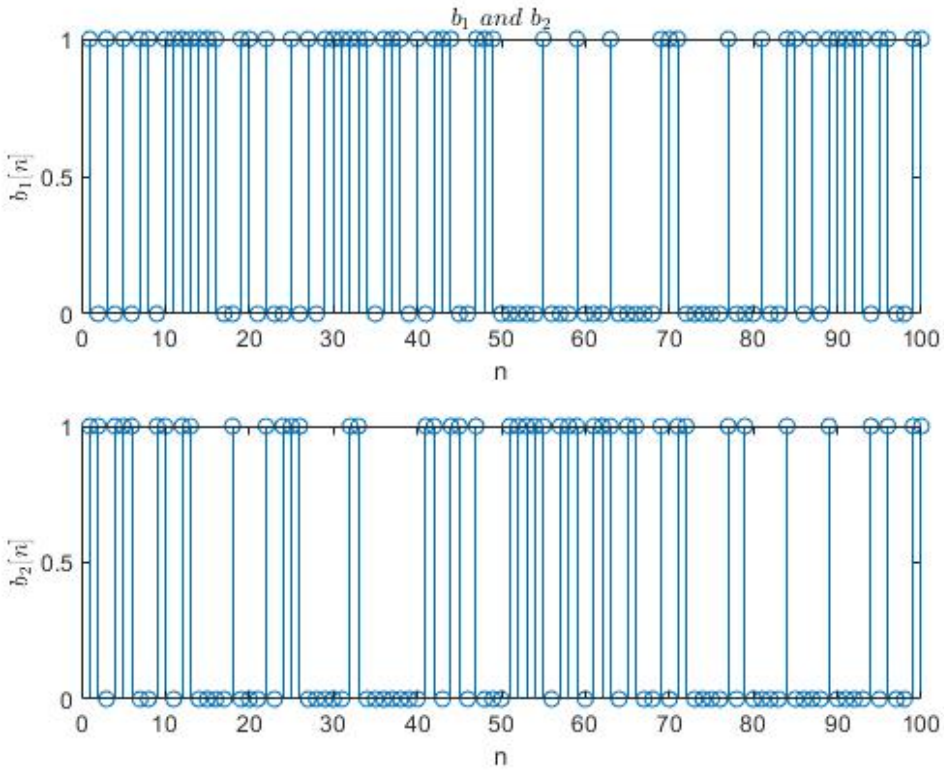
۵.۲: کلیت این قسمت مانند تابع AnalogMod میباشد و لازم است که سیگنال را در یک سری سینوس و کسینوس ضرب کنیم و خروجی را تولید کنیم. مشابه همان توضیحات قسمت قبل برای تولید فیلتر پایین گذر از یک تابع $sinc(t)$ تبدیل fft گرفته و سیگنال را از آن عبور میدهیم.

۶.۲: تابع اصلی این تابع است مطابق تعریف میدانیم که Filter Matched برای هرکدام از پالس های ما به صورت روبرو تعریف میشود: $MatchedFilter = h(t) = pulse(T-t)$ بنابر این با flip کردن شکل پالس مختص به بیت ۱ و ۰، MatchedFilter های مربوطه تولید میشوند. پس از آن باید سیگنالی که در ورودی دریافت کرده ایم را با MatchedFilter ها کانوالو کنیم و هر T ثانیه نمونه برداری کنیم. حال برای تصمیم گیری که هر نمونه نشان دهنده بیت ۱ بوده یا ۰ نیز کافی است که خروجی MatchedFilter متناظر با ۱ را با با خروجی متناظر با ۰ مقایسه کنیم و هرکدام بزرگتر بود، خروجی همان خواهد بود.

۳ انتقال دنباله تصادفی ۰ و ۱

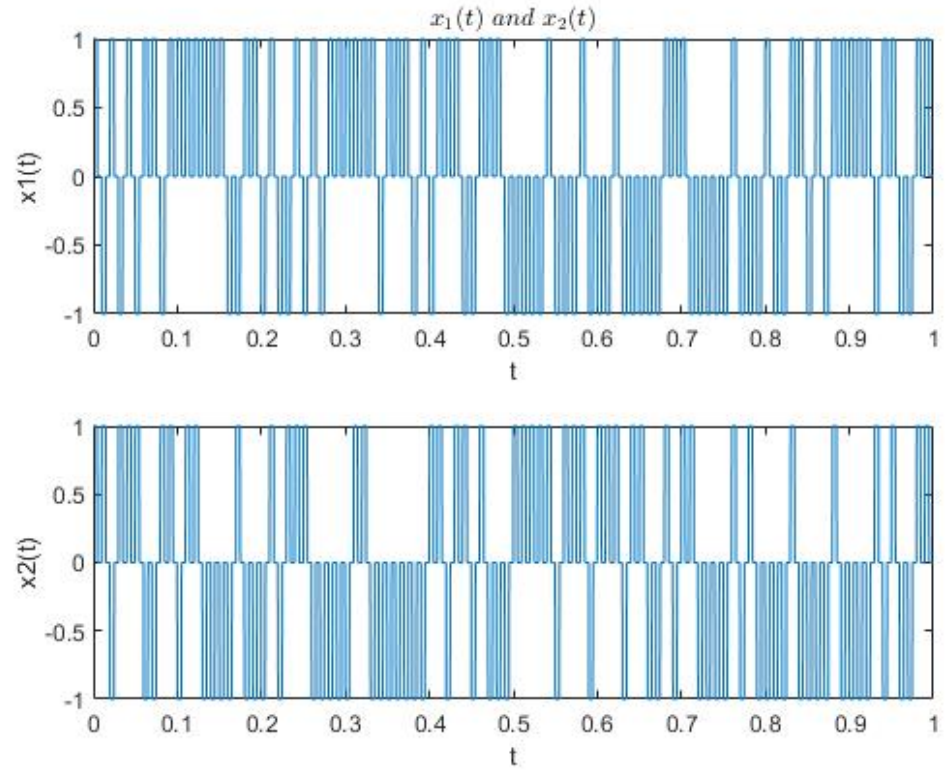
۱.۳: پارامتر های مساله را مطابق خواسته دستور پروژه تنظیم کردیم و موارد اضافه تری را که تنظیم کرده ایم در کد متلب توضیحاتش را کامنت کرده ایم.

آ:
خروجی های بلوک را به ترتیب نمایش میدهیم. شکل ها دارای ،Xlabel Title و Ylabel میباشند.

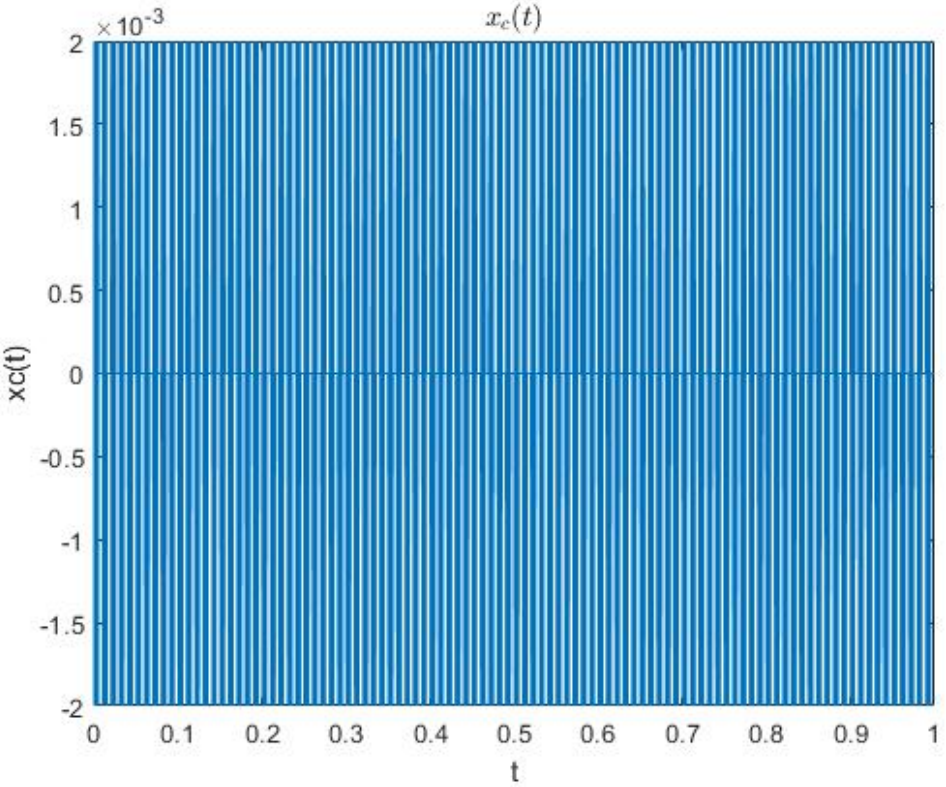


خروجی تابع Divide

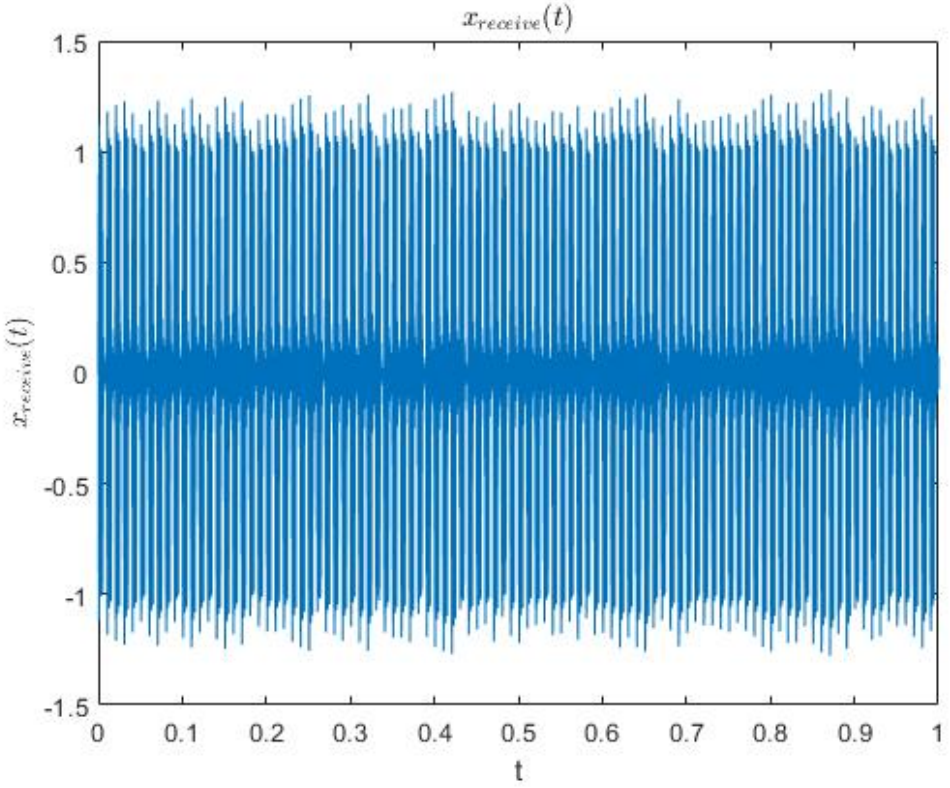
newline



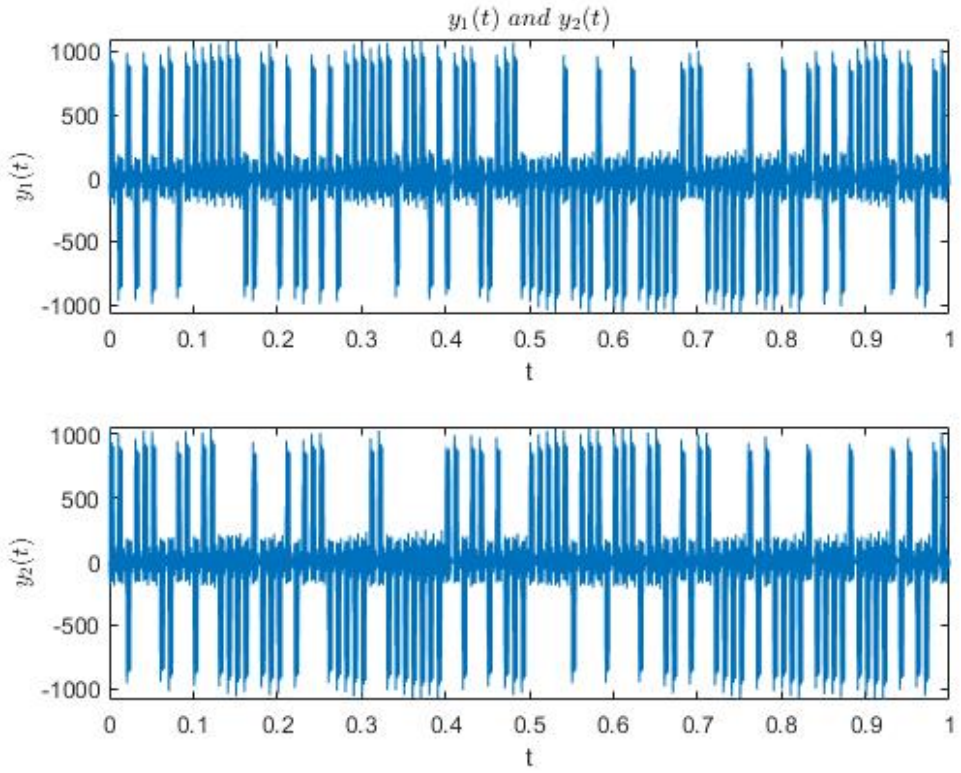
خروجی تابع PulseShaping



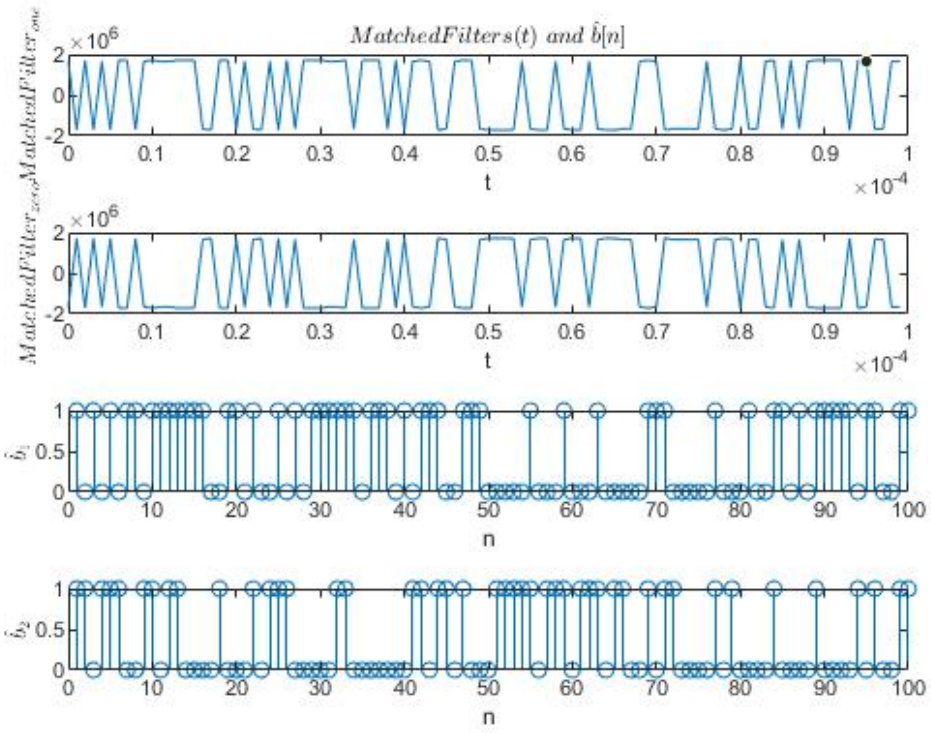
خروجی تابع AnalogMod



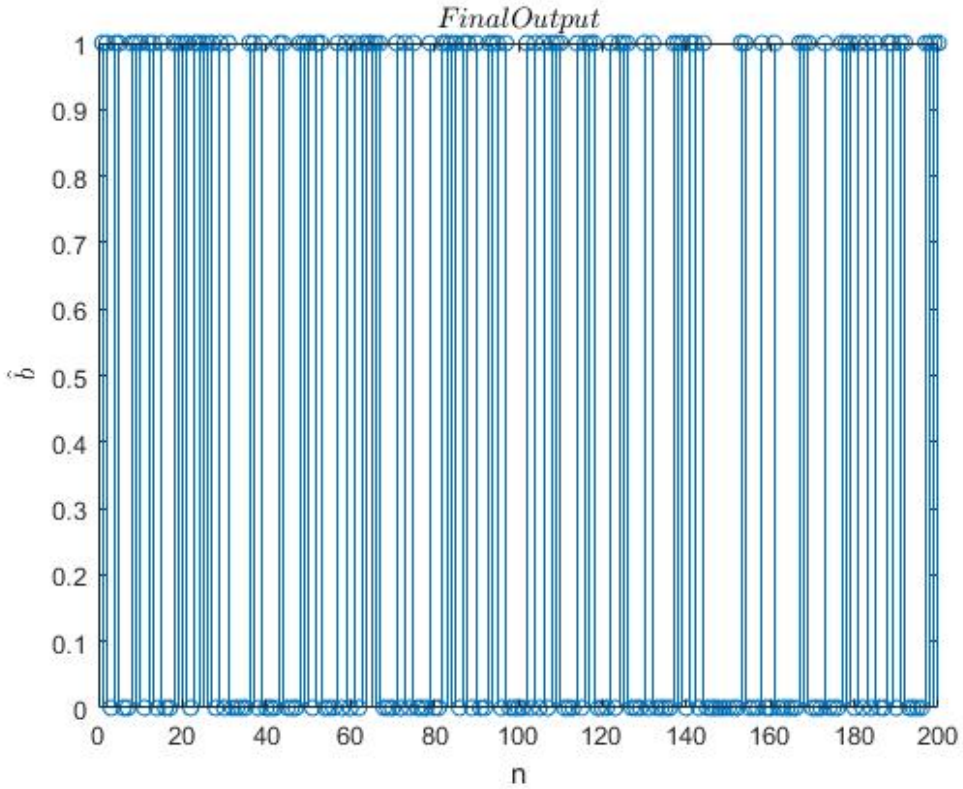
خروجی تابع Channel



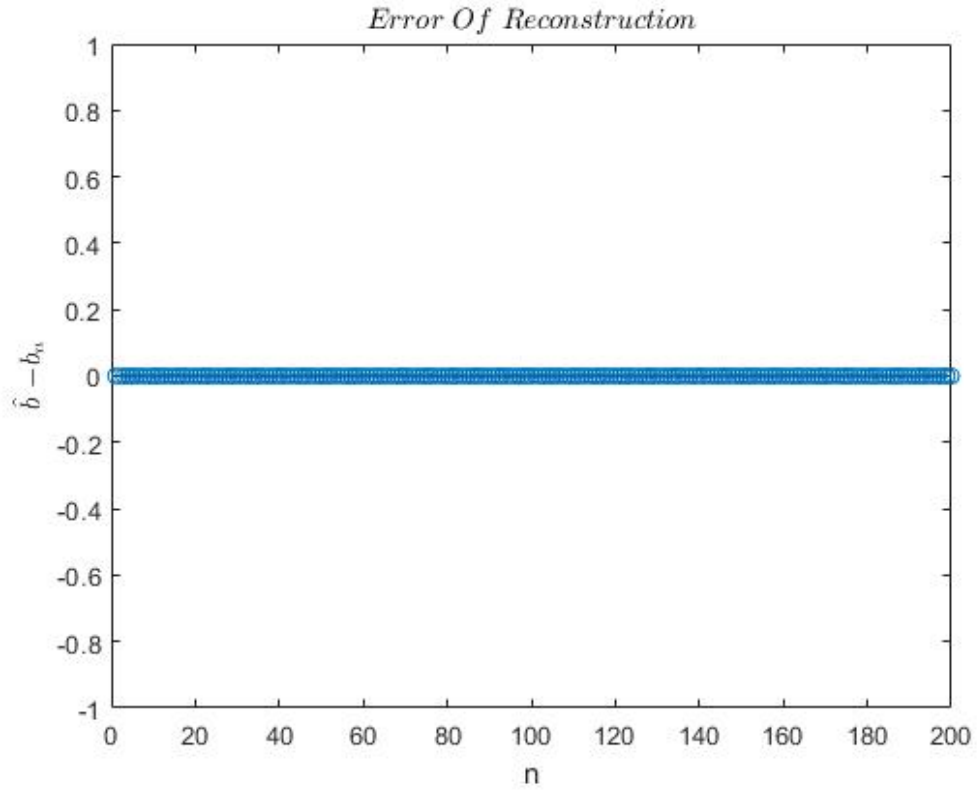
خروجی تابع AnalogDemod



خروجی تابع MatchedFilter

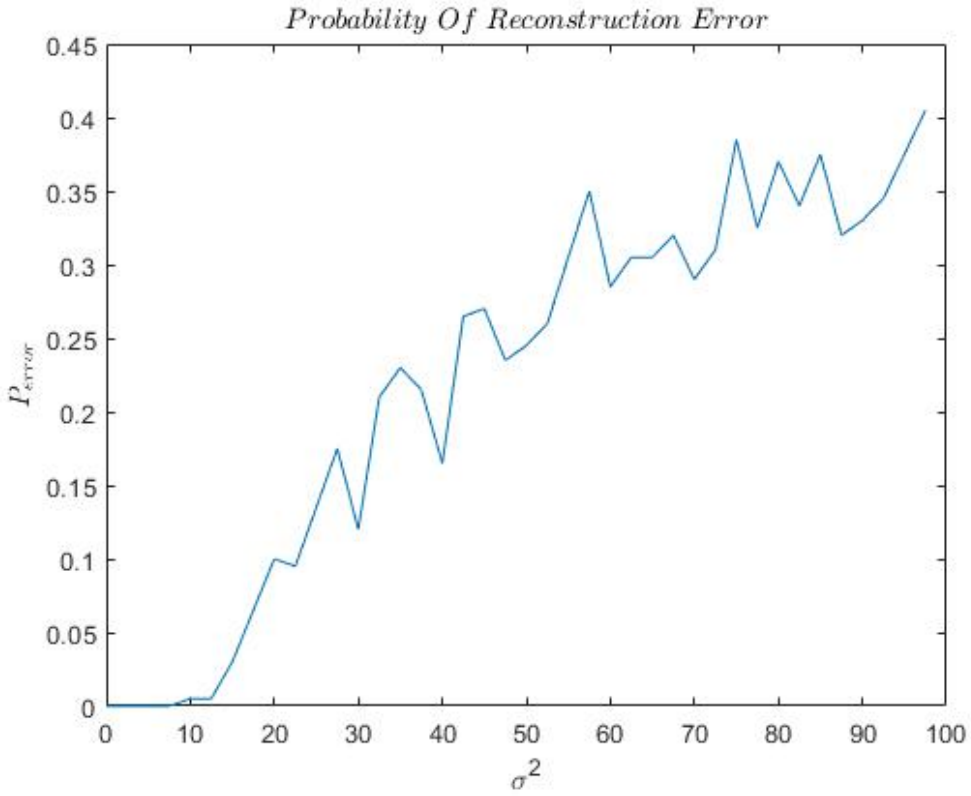


خروجی تابع Combine



خطای بازسازی

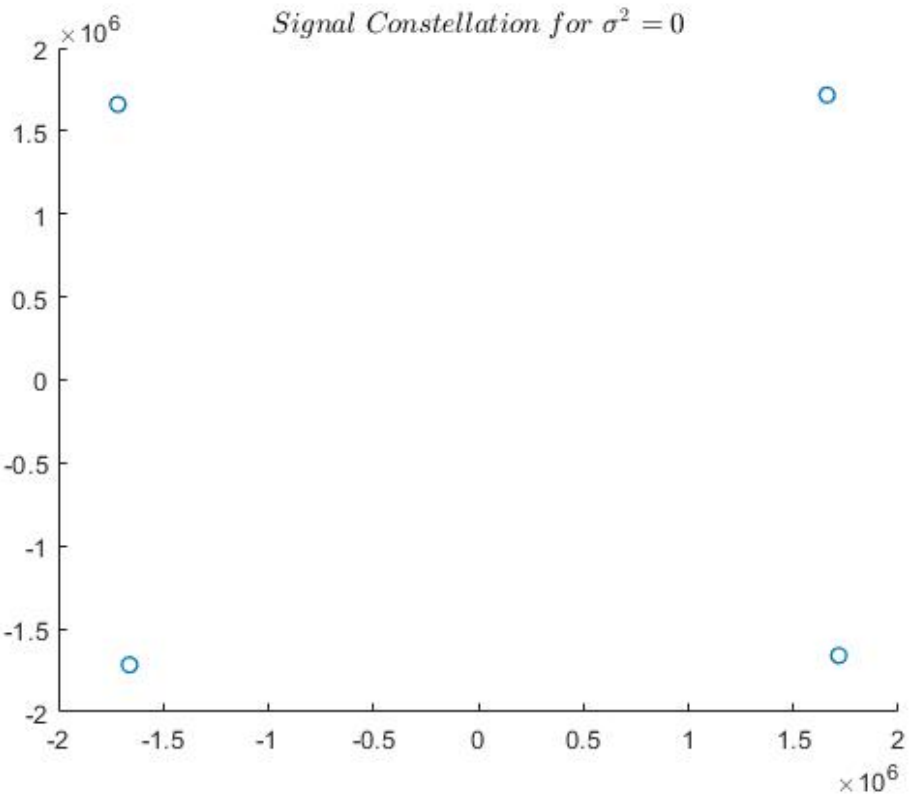
ب:
با استفاده از دستور normrnd در متلب که متغیر رندوم با توزیع گوسی و میانگین و واریانس دلخواه ما تولید میکند، به سیگنال خروجی از تابع Channel نویز می افزاییم. نتیجه خواسته شده به شرح زیر میباشد:



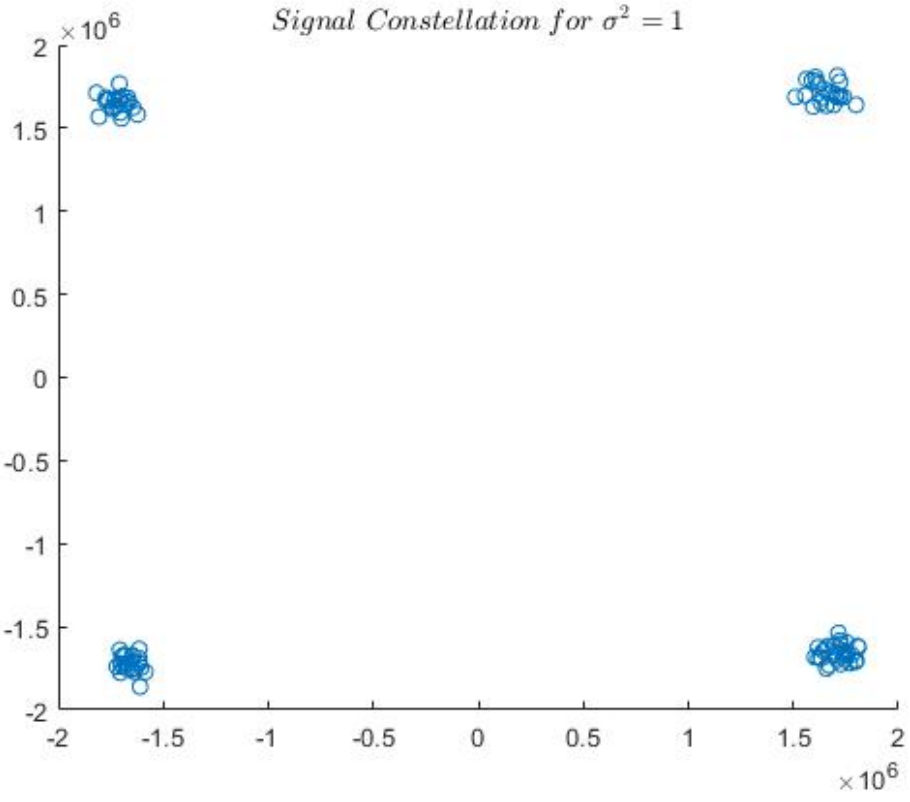
احتمال خطای بازسازی برحسب واریانس نویز کانال

واضح است که این رفتار کاملا مطابق با انتظار ماست. همانطور که در تصویر هم واضح است خطای باز سازی به ازای واریانس های کم نویز صفر است، یعنی دریافت کننده میتواند سیگنالی که آلوده به نویز کوچکی است را با دقت ۱۰۰ درصد بازسازی کند. اما رفته رفته با افزایش شدت نویز و افزایش واریانس آن، خطای بازسازی افزایش میابد و مطابق انتظارمان،، وقتی که واریانس نویز بیشتر و بیشتر میشود خطای ما به ۵۰ درصد میل میکند چون اعداد رندوم تولید کرده که یا با بیت ارسال شده برابر یا نابرابر است بنابر این احتمال خطا به ۵۰ درصد میل میکند.
لازم به ذکر است رفتار کمی غیرخطی شکل فوق به این خاطر است که نمودار با ۴۰ نمونه ترسیم شده تا زمان زیادی برای پردازش صرف نشود.

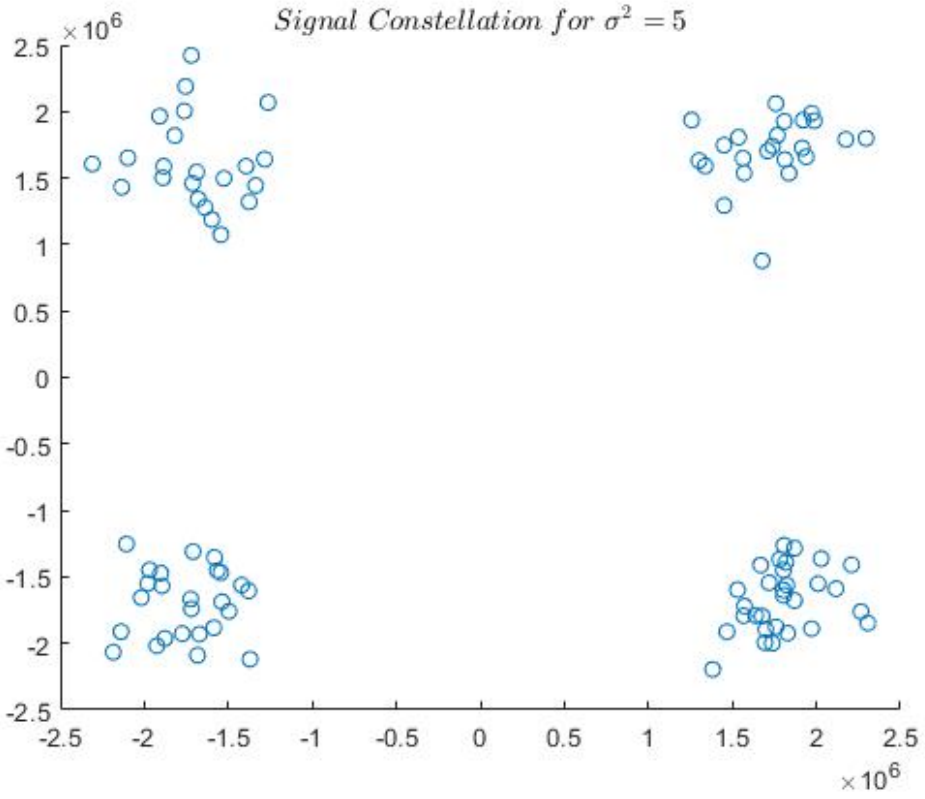
ج:
برای این قسمت از خروجی دوم تابع MatchedFilter استفاده کرده ایم. شکل های زیر Signal Constellation میباشند که بر اساس تعریفی که در متن پروژه است رسم شده اند. این را به ازای ۶ مقدار مختلف واریانس نویز رسم کرده ایم که نتیجه مانند زیر میباشد:



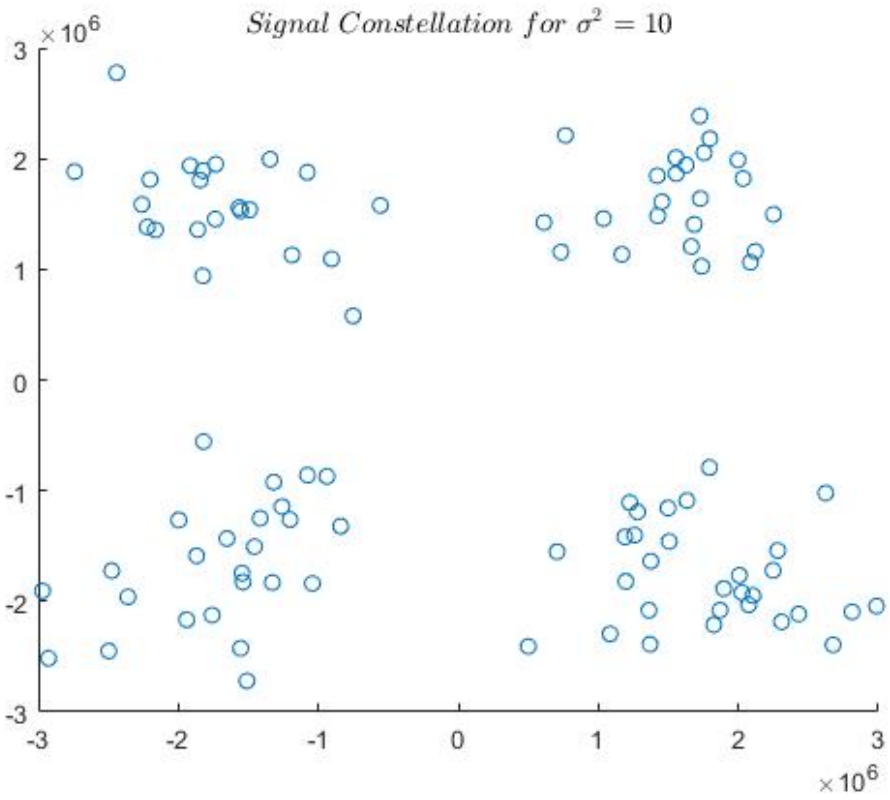
نویز با واریانس $\sigma^2 = 0$



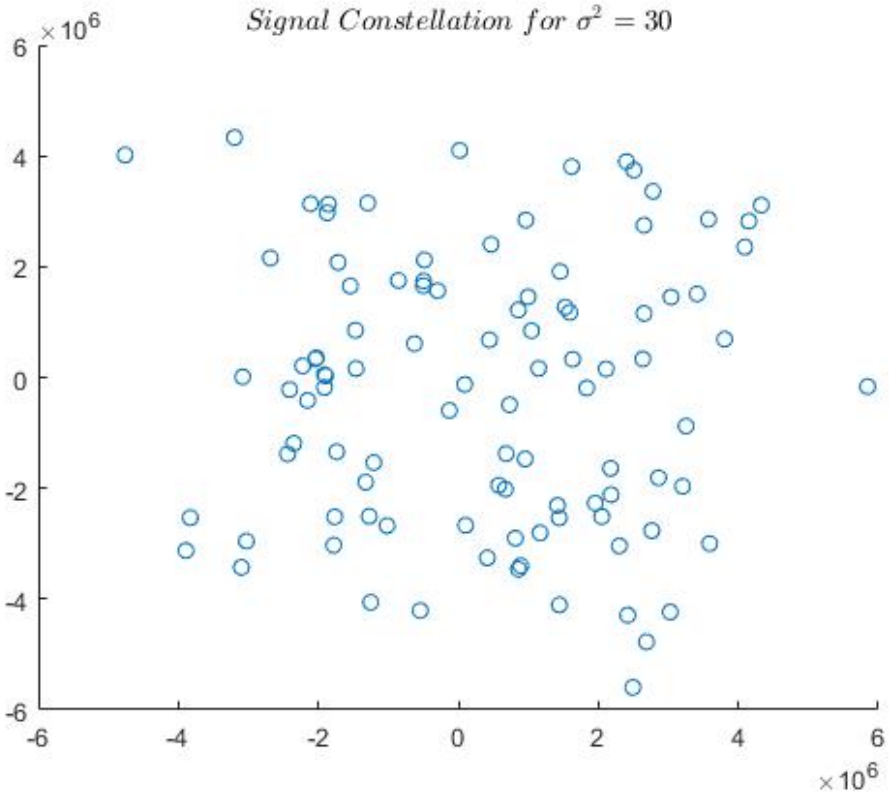
نویز با واریانس $\sigma^2 = ۱$



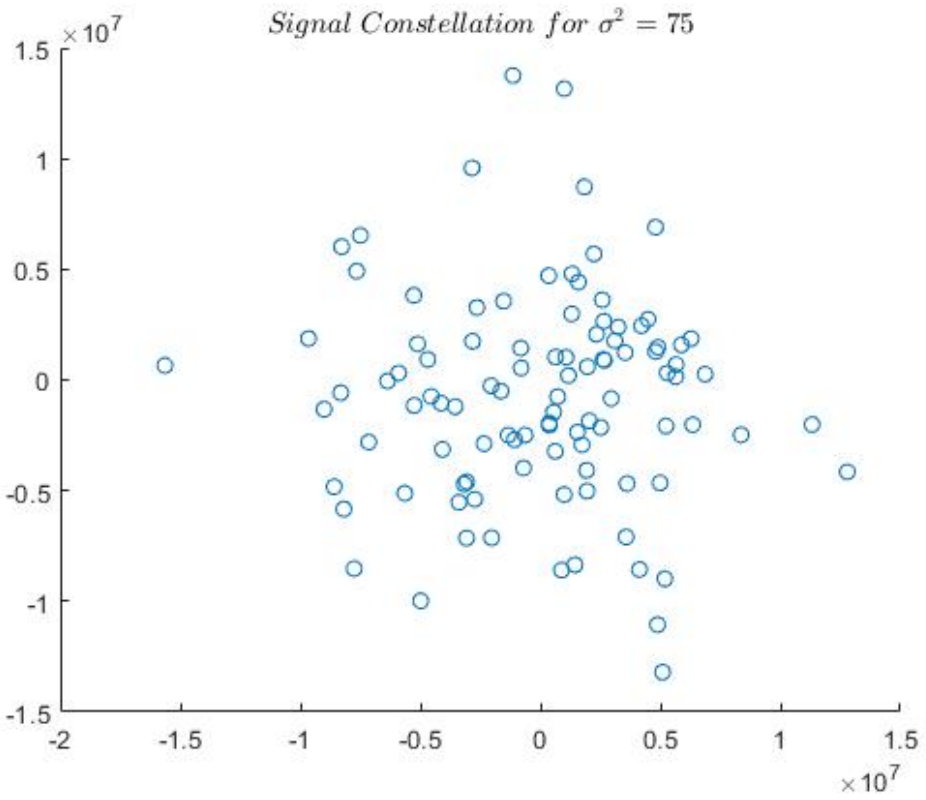
نویز با واریانس $\sigma^2 = ۵$



نویز با واریانس $\sigma^2 = ۱۰$



نویز با واریانس $\sigma^2 = ۳۰$

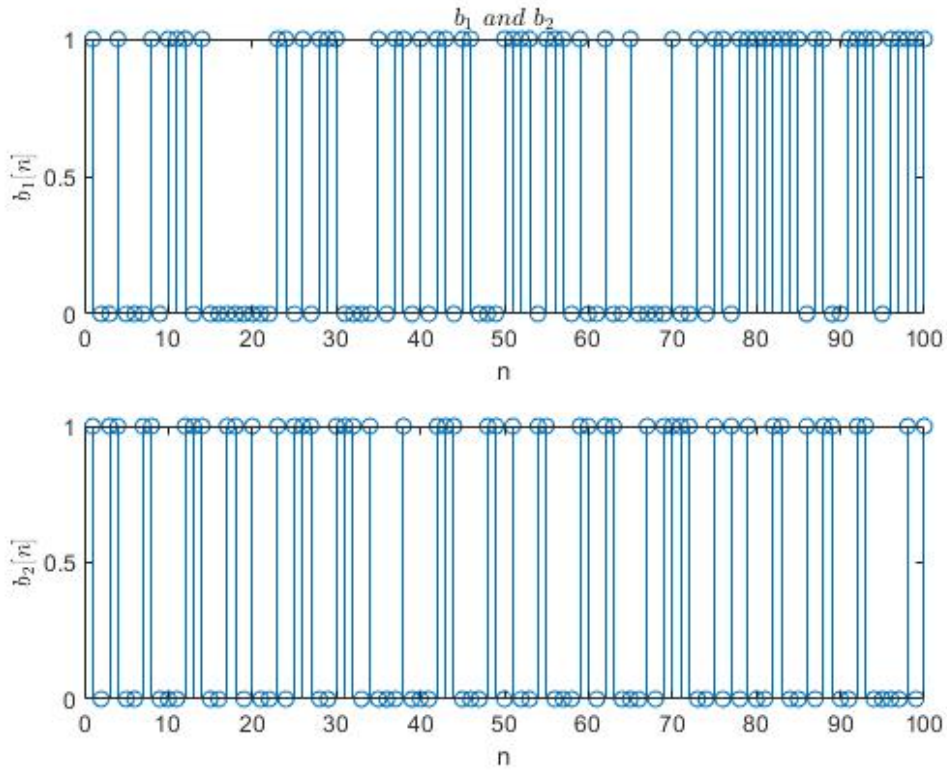


نویز با واریانس $\sigma^2 = ۷۵$

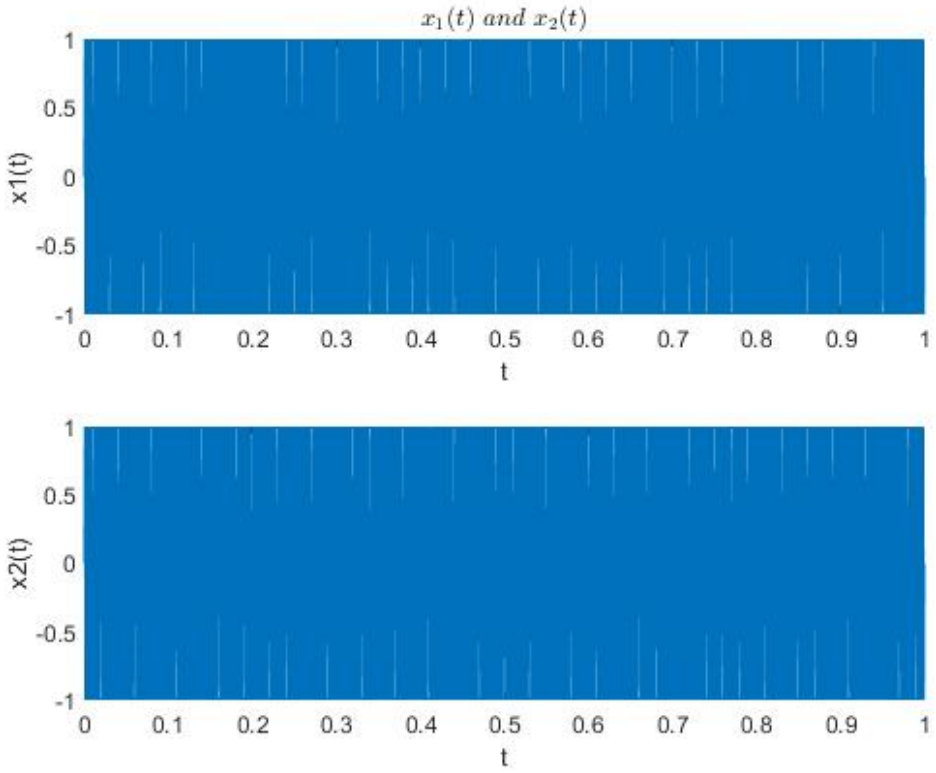
واضح است مطابق انتظار، هرچه نویز کمتر باشد تراکم این اعداد بازیابی شده باید بیشتر باشد. تا جایی که به ازای $\sigma^2 = ۰$ نویز کاملا بر هم منطبق هستند. رفته رفته با افزایش واریانس نویز این نقاط از هم فاصله میگیرند تا جایی که به نظر میرسد برای واریانس های بسیار بالا اصلا هیچ تراکمی در منطقه خاصی وجود ندارد.

۲.۳: کلیت این بخش و بخش قبل شباهت بسیار زیادی دارد. لذا از توضیحات تکراری پرهیز میکنیم و نتایج را به نمایش میگذاریم.

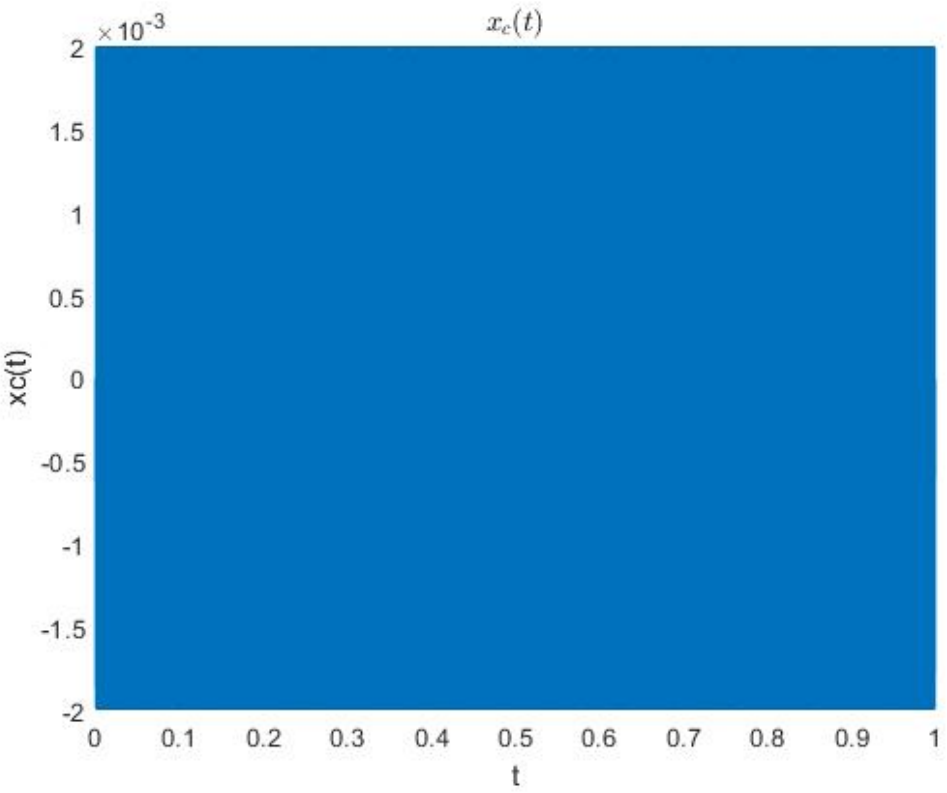
آ:
خروجی های بلوک را به ترتیب نمایش میدهیم. شکل ها دارای Xlabel ،Title و Ylabel میباشند.



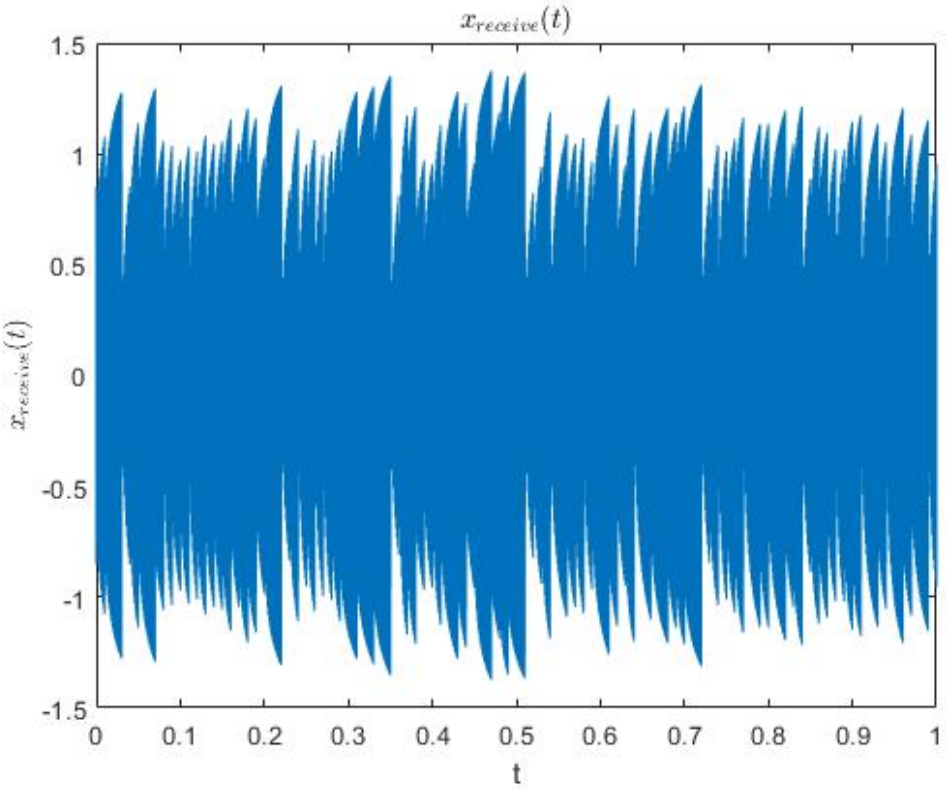
خروجی تابع Divide



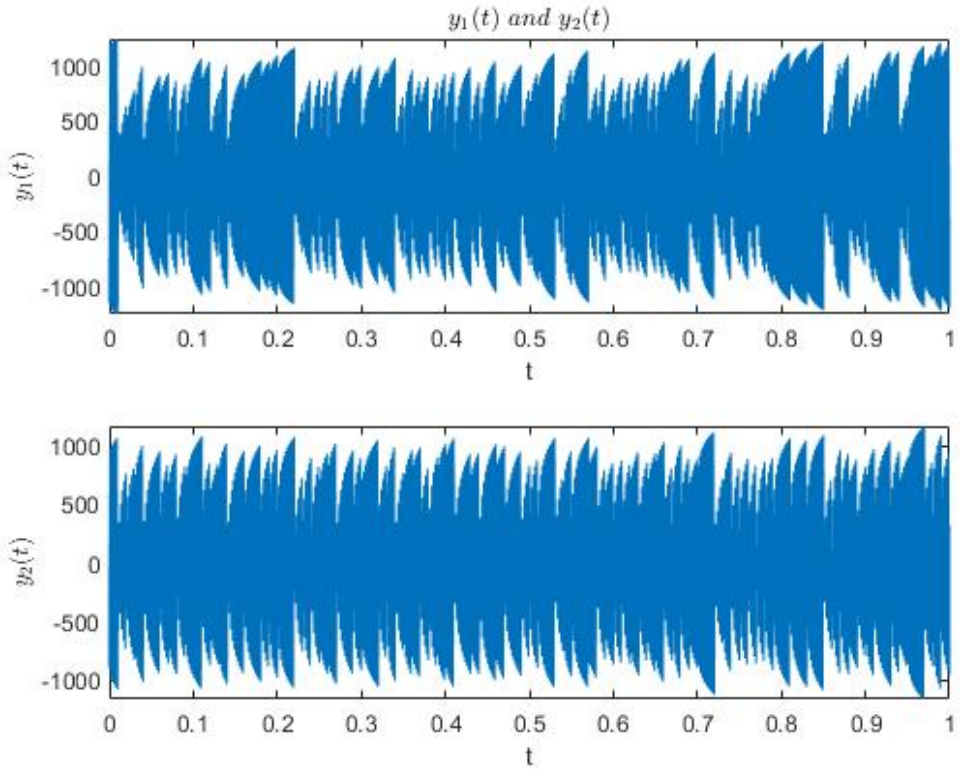
خروجی تابع PulseShaping



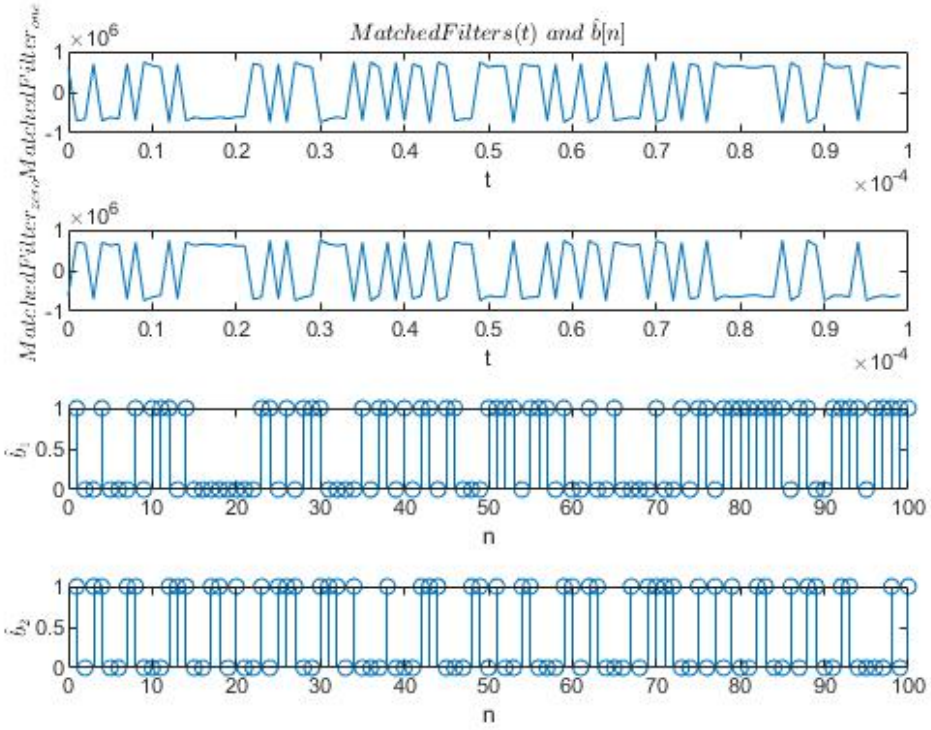
خروجی تابع AnalogMod



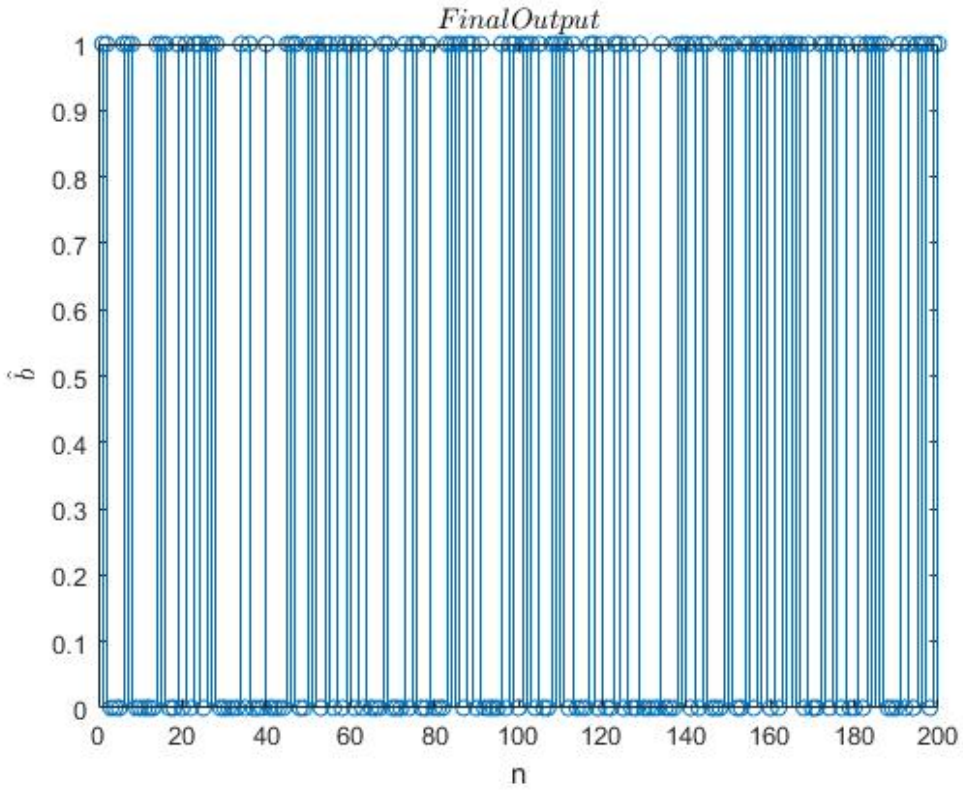
خروجی تابع Channel



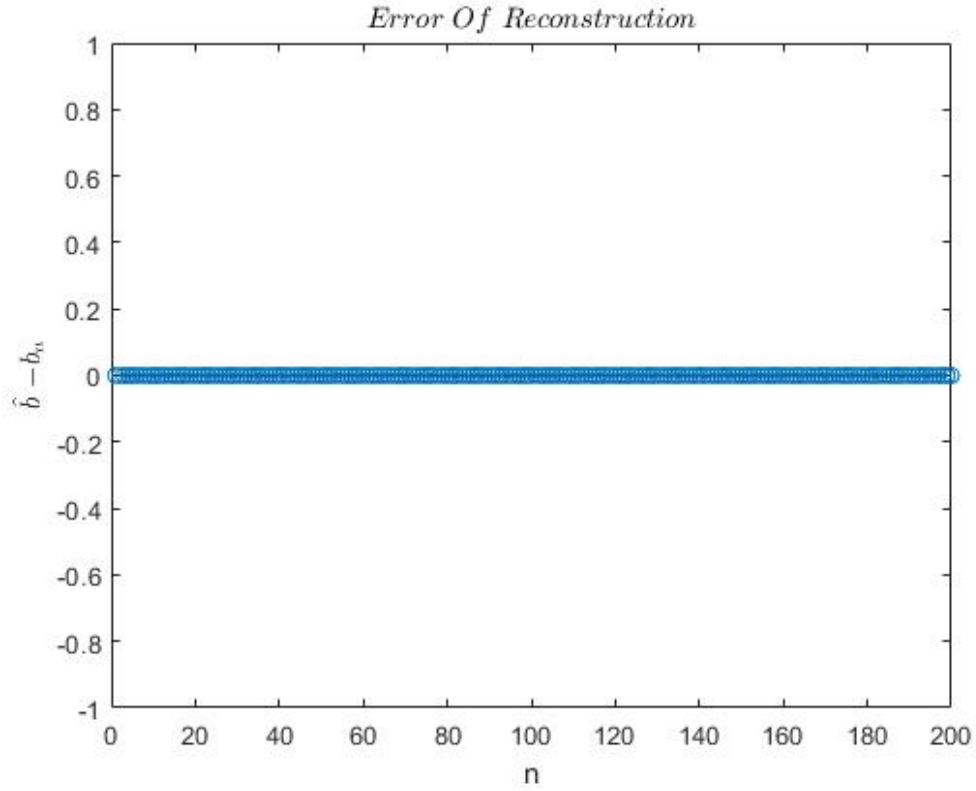
خروجی تابع AnalogDemod



خروجی تابع MatchedFilter

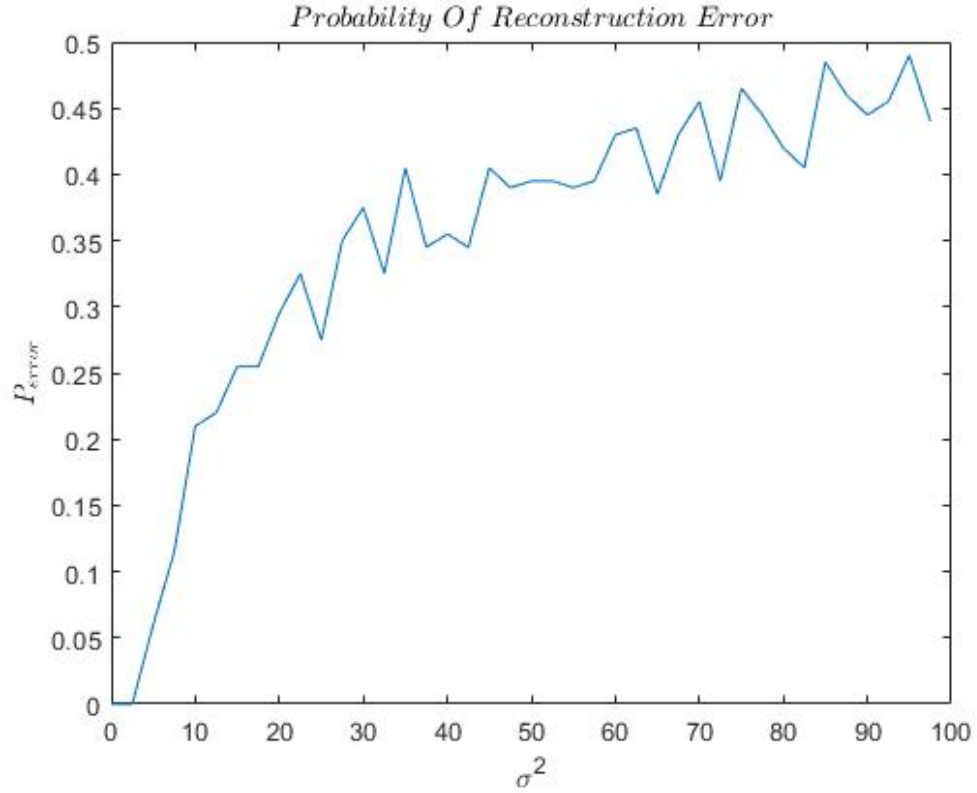


خروجی تابع Combine



خطای بازسازی

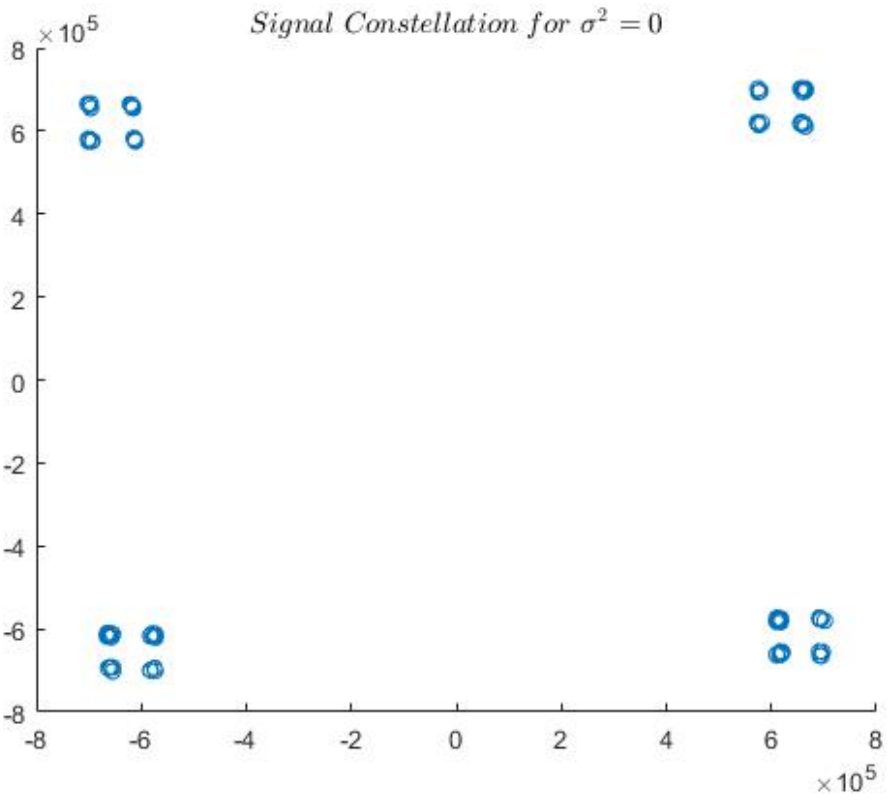
ب:
با استفاده از دستور normrnd در متلب که متغیر رندوم با توزیع گوسی و میانگین و واریانس دلخواه ما تولید میکند، به سیگنال خروجی از تابع Channel نویز می افزاییم. نتیجه خواسته شده به شرح زیر میباشد:



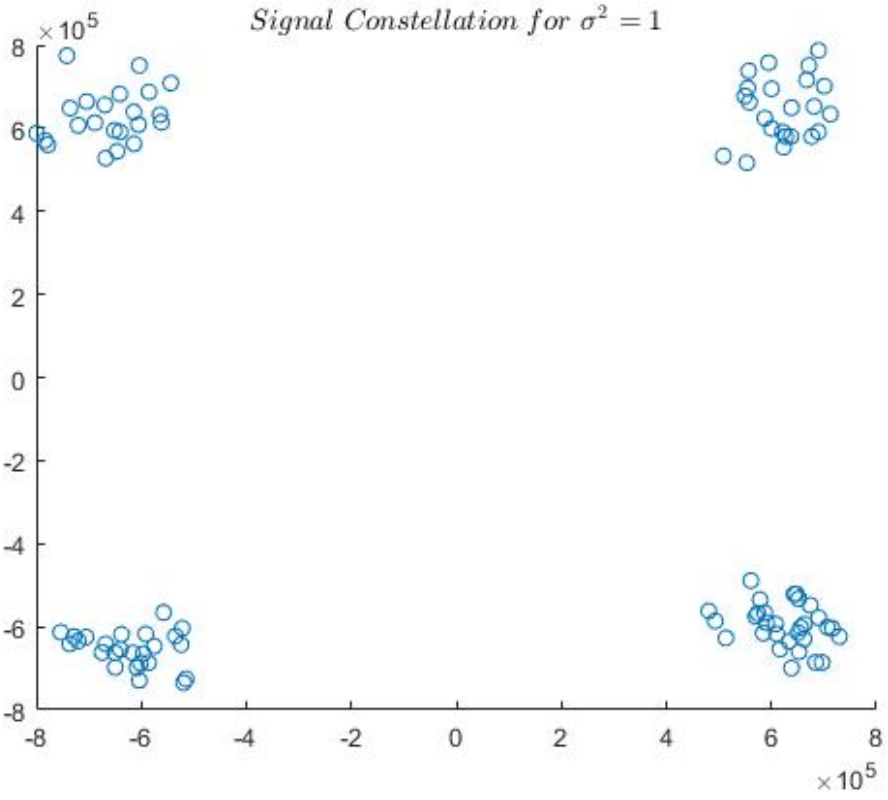
احتمال خطای بازسازی برحسب واریانس نویز کانال

واضح است انتظار داشتیم نتیجه ای مشابه با قسمت قبل را شاهد باشیم.البته تفاوت های ریزی نیز وجود دارد که در شکل واضح است. اما نکته مهم این است که همچنان مجموعهپ ما میتواند تا حدی اثر نویز را خنثی کند و بدون هیچ اشتباهی سیگنال را بازیابی کند. اما رفته رفته احتمال خطا بیشتر میشود تا اینکه اصلا به نزدیکی خطای ۵۰ درصد میرسیم که یعنی اصلا یک سیگنال رندوم جدید تولید شده است (مشابه با علتی که در قسمت قبل توضیح دادیم)
لازم به ذکر است رفتار کمی غیرخطی شکل فوق به این خاطر است که نمودار با ۴۰ نمونه ترسیم شده تا زمان زیادی برای پردازش صرف نشود.

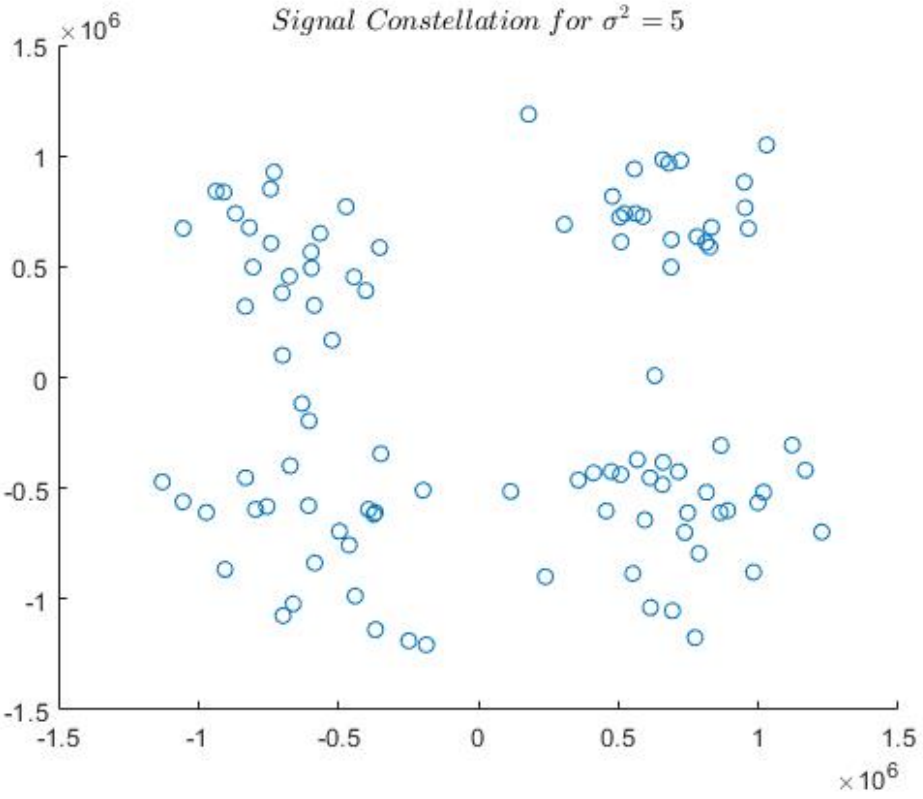
ج:
بدون تکرار توضیحات مشابه قبل، نتایج شبیه سازی را با هم میبینیم:



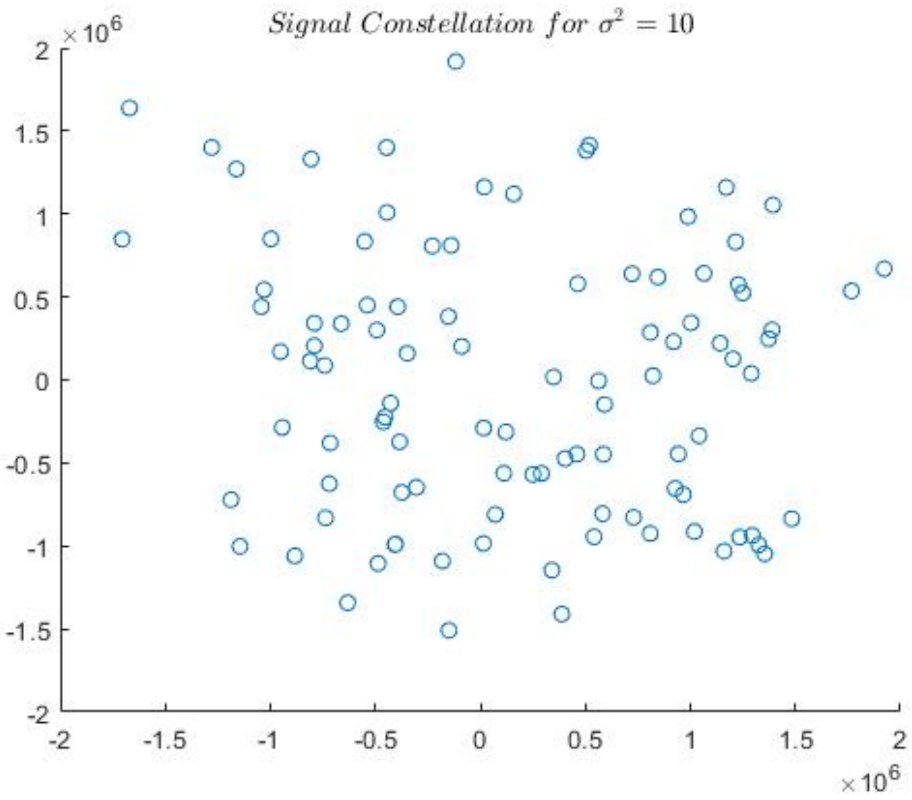
نویز با واریانس $\sigma^2 = 0$



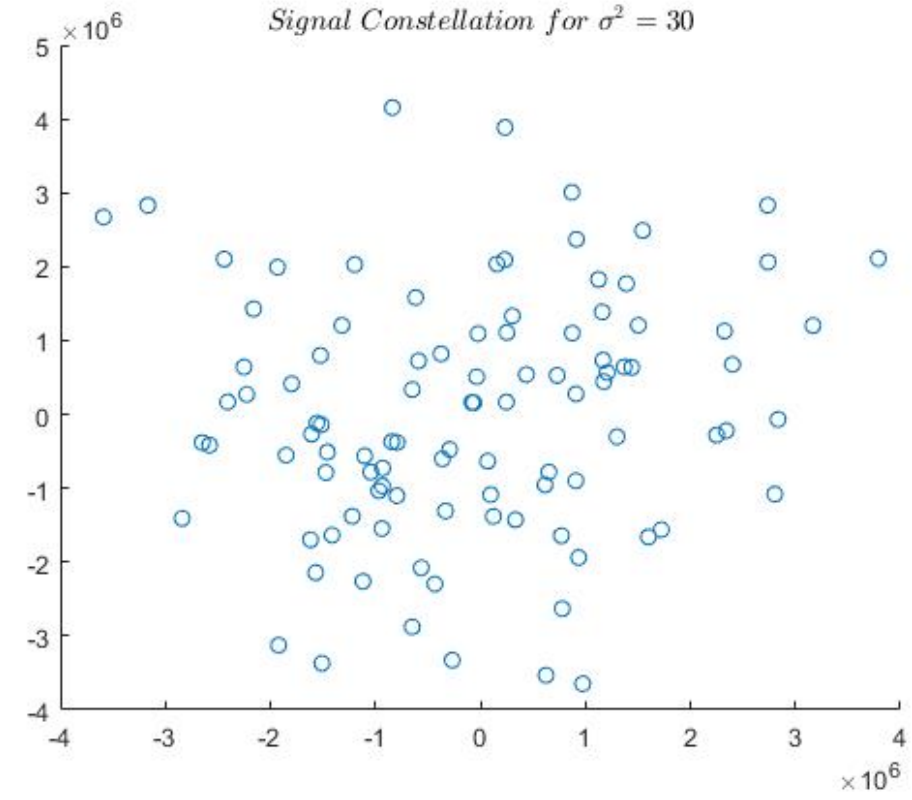
نویز با واریانس $\sigma^2 = ۱$



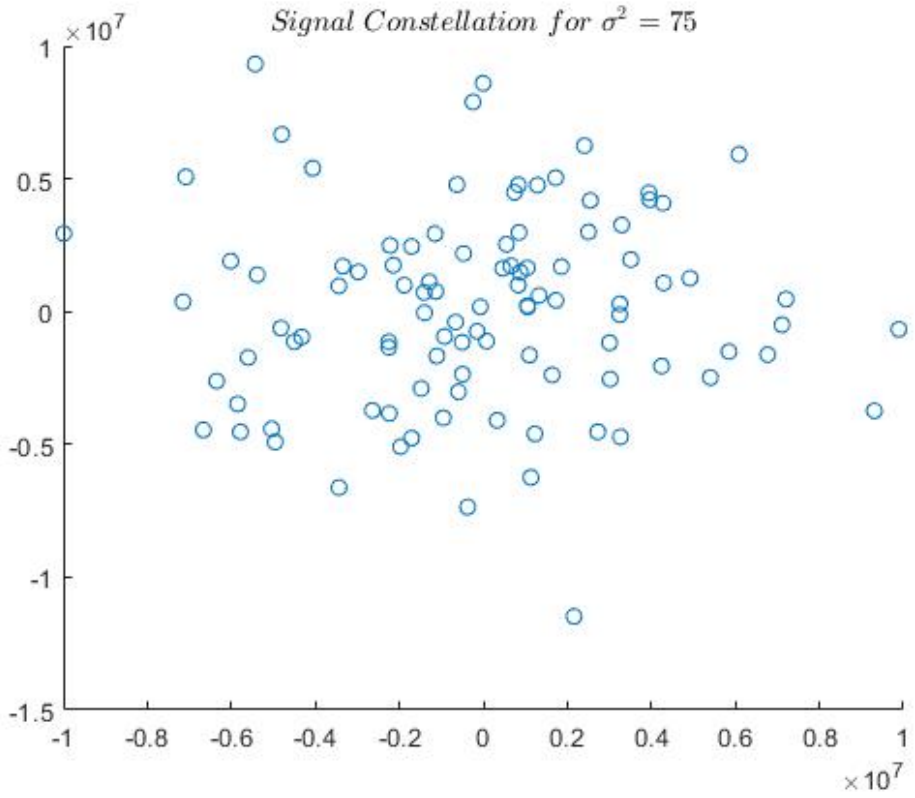
نویز با واریانس $\sigma^2 = ۵$



نویز با واریانس $\sigma^2 = ۱۰$



نویز با واریانس $\sigma^2 = ۳۰$



نویز با واریانس $\sigma^۲ = ۷۵$

دو نکته مهم قابل بیان است. یکی اینکه با مقایسه با نتیجه این قسمت که در حالت PSK انجام شده با نتیجه قسمت قبل که حالت FSK بود، واضح است این روش نسبت به نویز کمی حساس تر است و زودتر منظومه سیگنال ها تراکم خودشان را از دست میدهند.

نکته دوم نیز منظومه سیگنال ترسیم شده برای حالت $\sigma^۲_{noise} = ۰$ است که علی رغم اینکه تمام نقاط روی هم ترسیم نشده اند، اما همچنان از شکل میتوان استنباط کرد که فرایند بازسازی بدون خطا بوده است. علت اینکه در هر ناحیه بجای تمرکز بر یک نقطه، تمرکز بر چهار قسمت است این است که سیگنال های متناظر با بیت های ما سینوسی هستند. از طرفی فیلتر های ما با توجه به سبکی که تعریفشان کرده ایم ایده آل نیستند و برای همین در ابتدای سیگنال کمی دامنه آن ها کمتر است. به نوعی میتوان گفت سیگنال های خروجی از فیلتر ها یک بخش transient دارند و یک بخش steady state که بخش گذرا باعث بوجود آمدن این پدیده شده است.

۳.۳: کلیت این بخش و بخش قبل شباهت بسیار زیادی دارد. لذا از توضیحات تکراری پرهیز میکنیم و نتایج را به نمایش میگذاریم.

آ:

بله، دو سیگنال متعامدند باید ثابت کنیم که انتگرال حاصلضرب دو سیگنال روی یک دوره تناوب حاصلی برابر با صفر دارد:

$$\int_{۰}^{T_s} sin(۲\pi\times ۱۵۰۰\times t)sin(۲\pi\times ۱۰۰۰\times t)\,dt = \frac{۱}{۲}\int_{۰}^{T_s}[cos(۲\pi\times ۵۰۰\times t)-cos(۲\pi\times ۲۵۰۰\times t)]\,dt$$

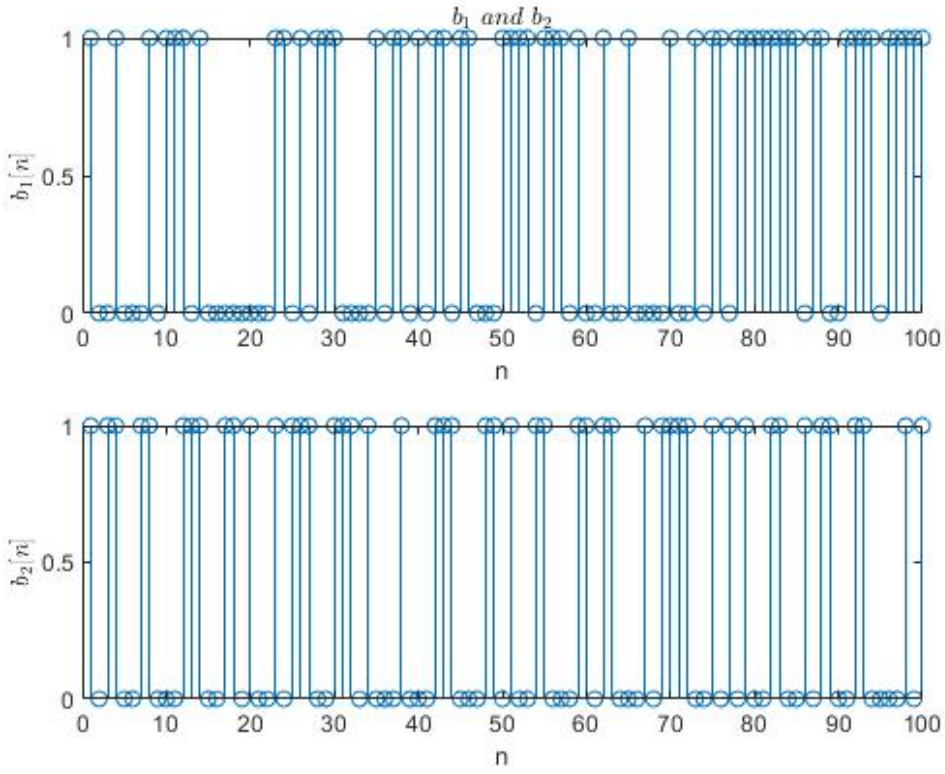
$$= \frac{۱}{۲}T_s \frac{sin(۲\pi \times ۵۰۰ \times T_s)}{۲\pi \times ۵۰۰ \times T_s} - \frac{۱}{۲}T_s \frac{sin(۲\pi \times ۲۵۰۰ \times T_s)}{۲\pi \times ۲۵۰۰ \times T_s}$$

و از آنجا که $T_s = integer$ بنابراین هر دو سینوس برابر صفر خواهند بود. لذا:

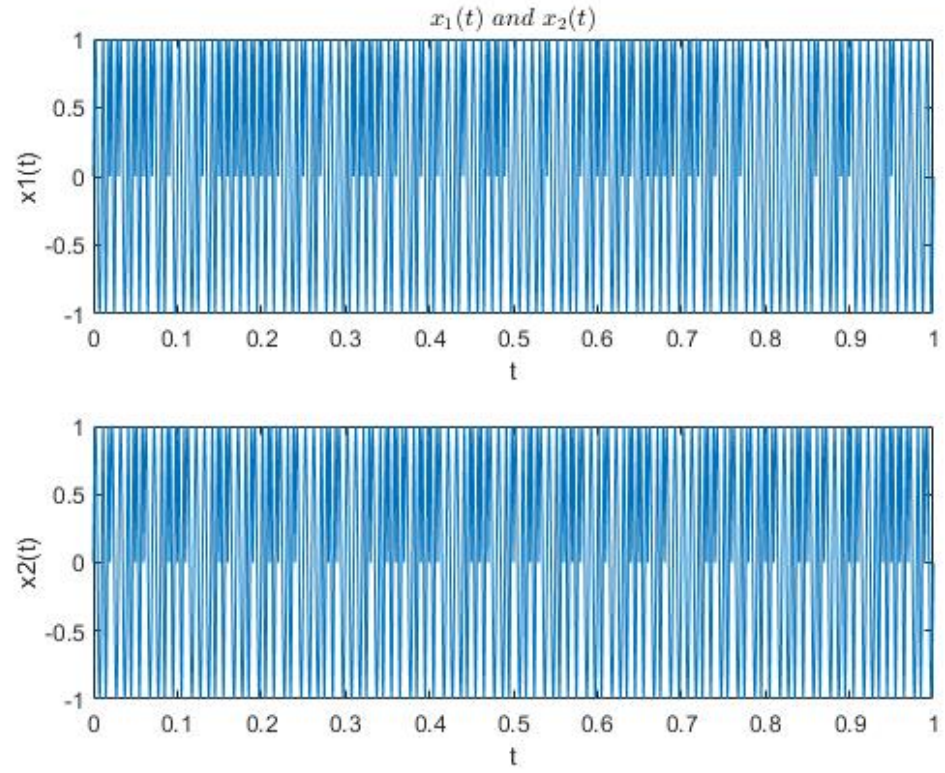
$$\int_{۰}^{T_s} sin(۲\pi \times ۱۵۰۰ \times t) sin(۲\pi \times ۱۰۰۰ \times t)\, dt = ۰$$

اثبات شد.

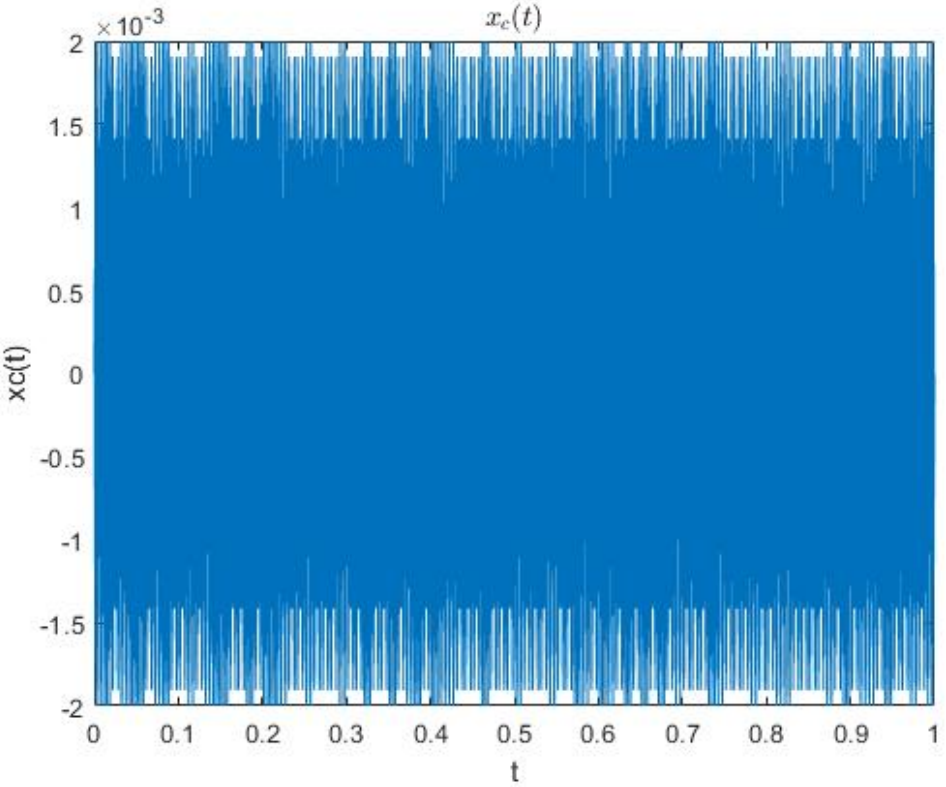
ب:
خروجی های بلوک را به ترتیب نمایش میدهیم. شکل ها دارای Xlabel Title، Ylabel و میباشند.



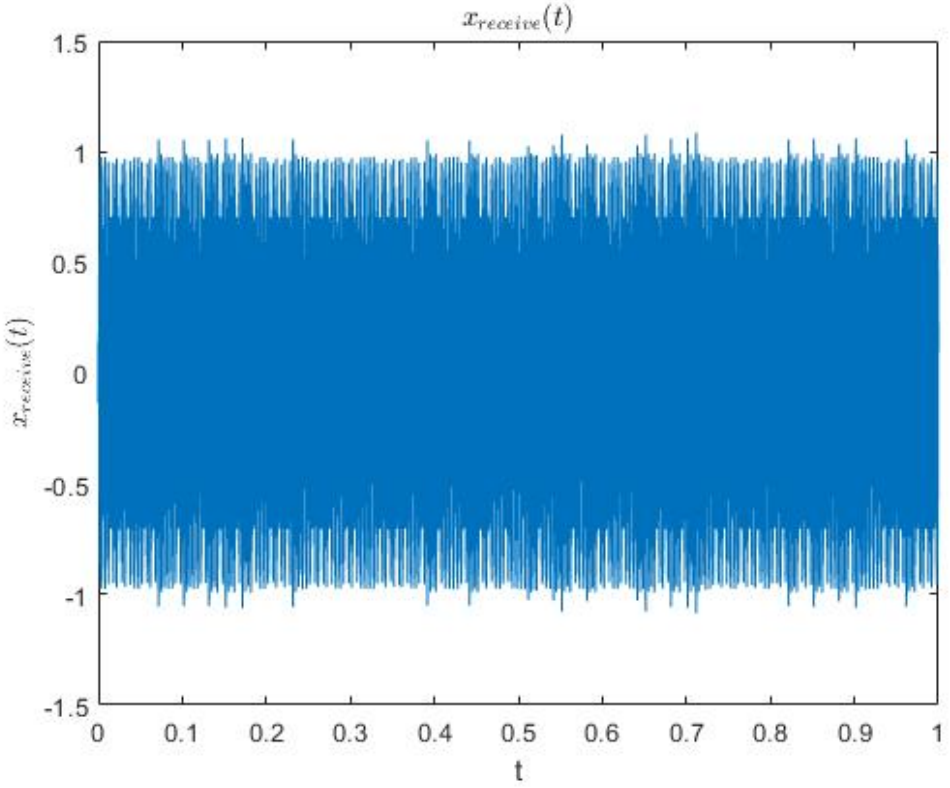
خروجی تابع Divide



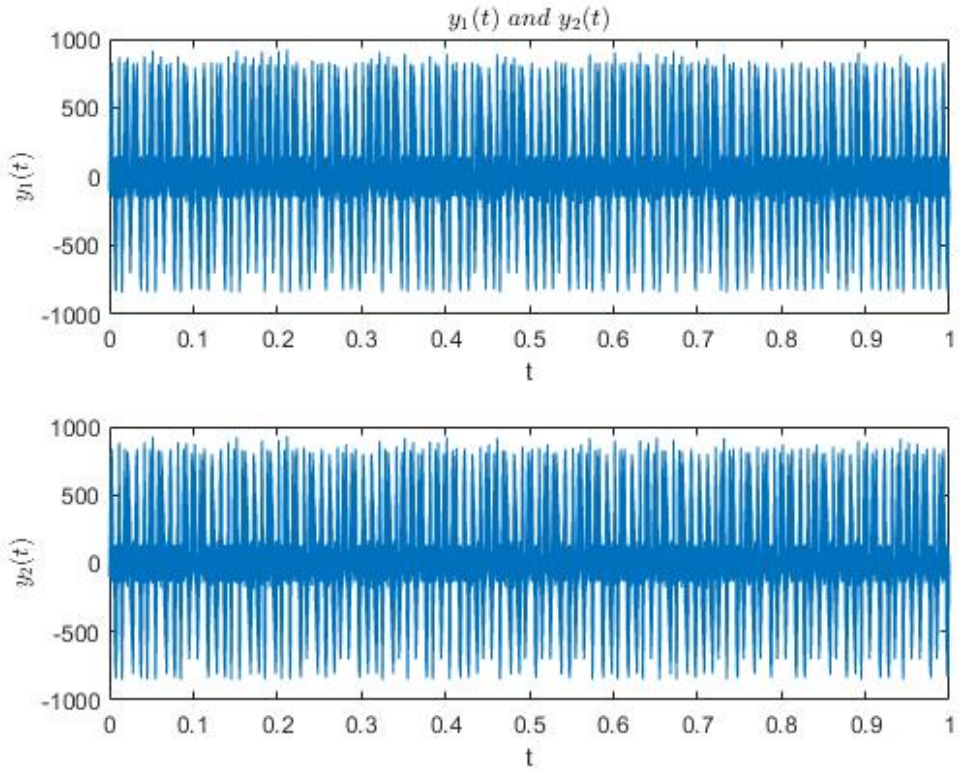
خروجی تابع PulseShaping



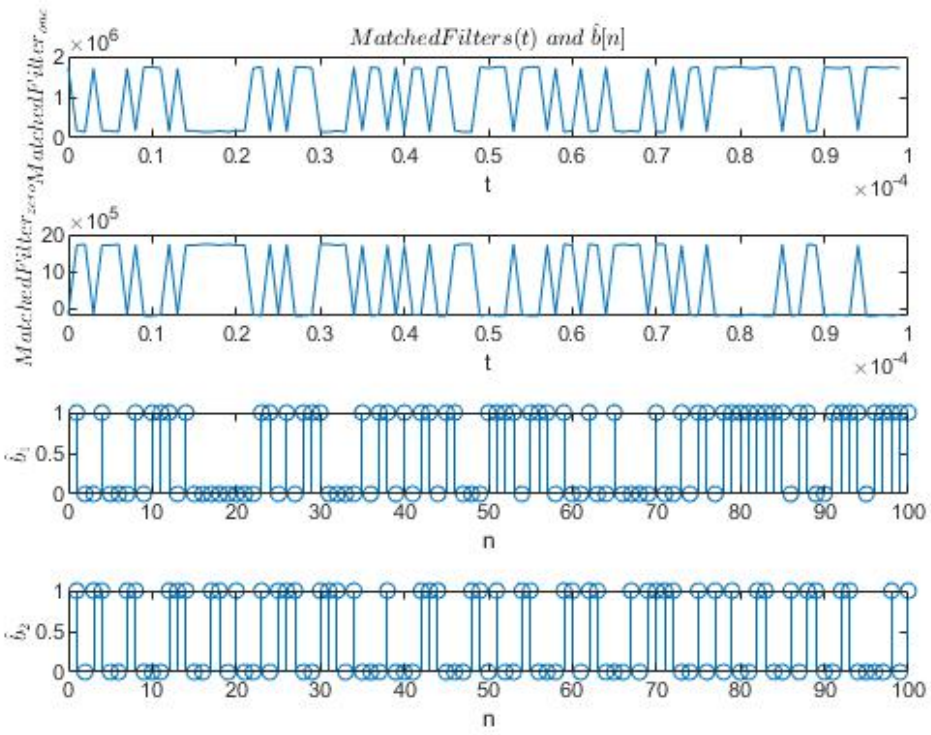
خروجی تابع AnalogMod



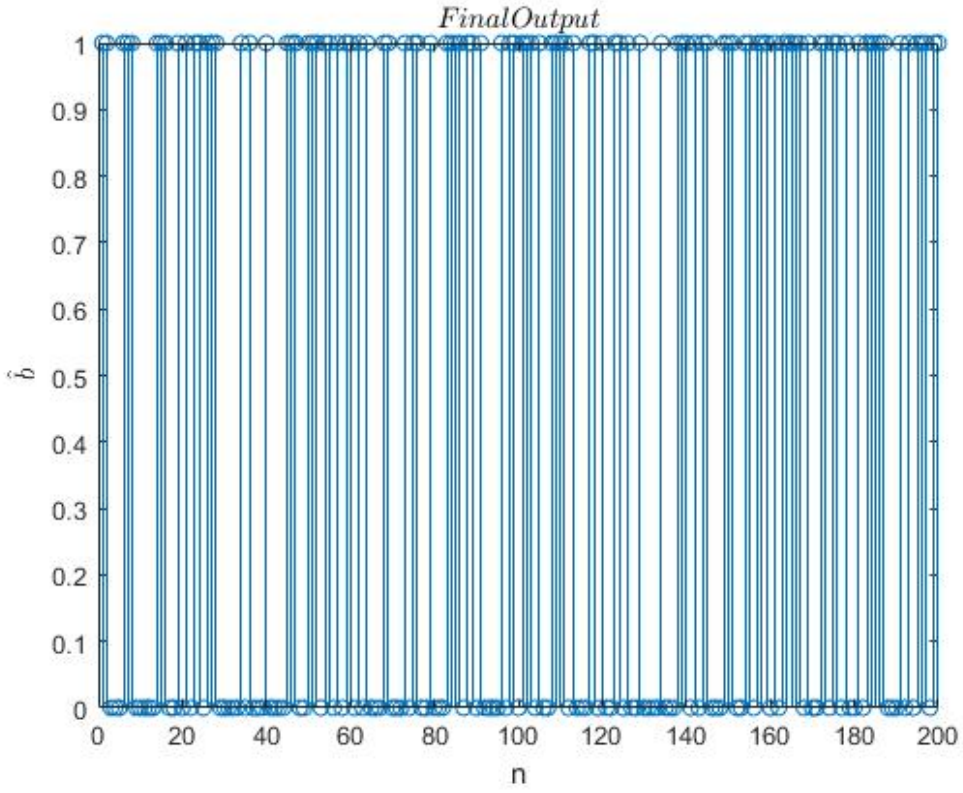
خروجی تابع Channel



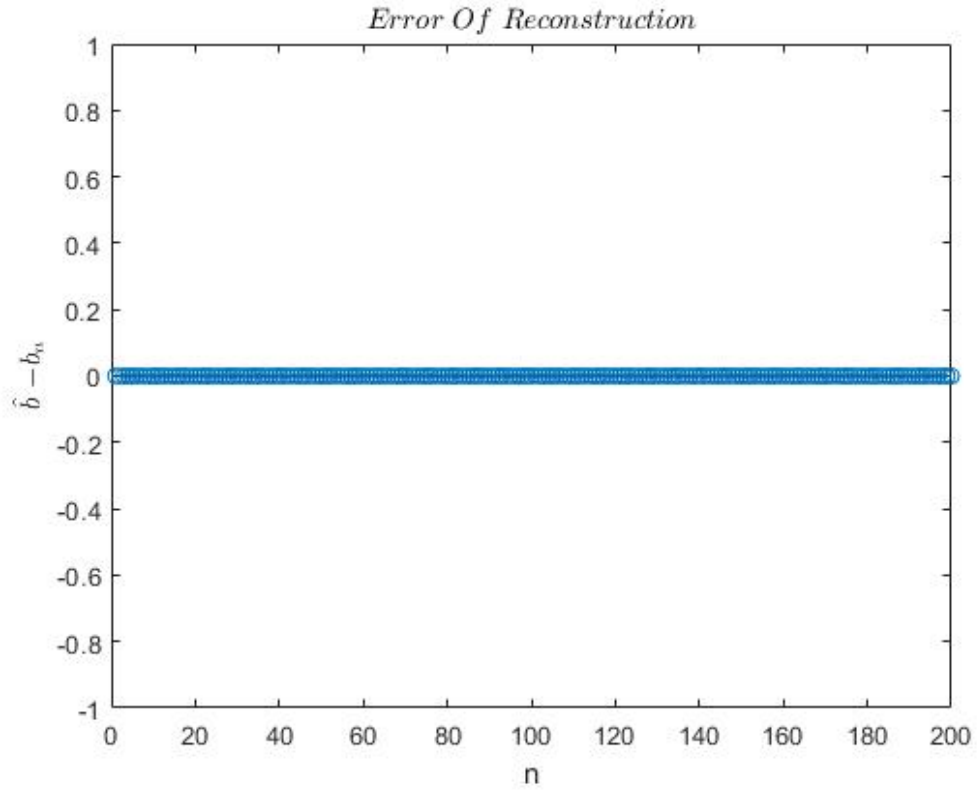
خروجی تابع AnalogDemod



خروجی تابع MatchedFilter



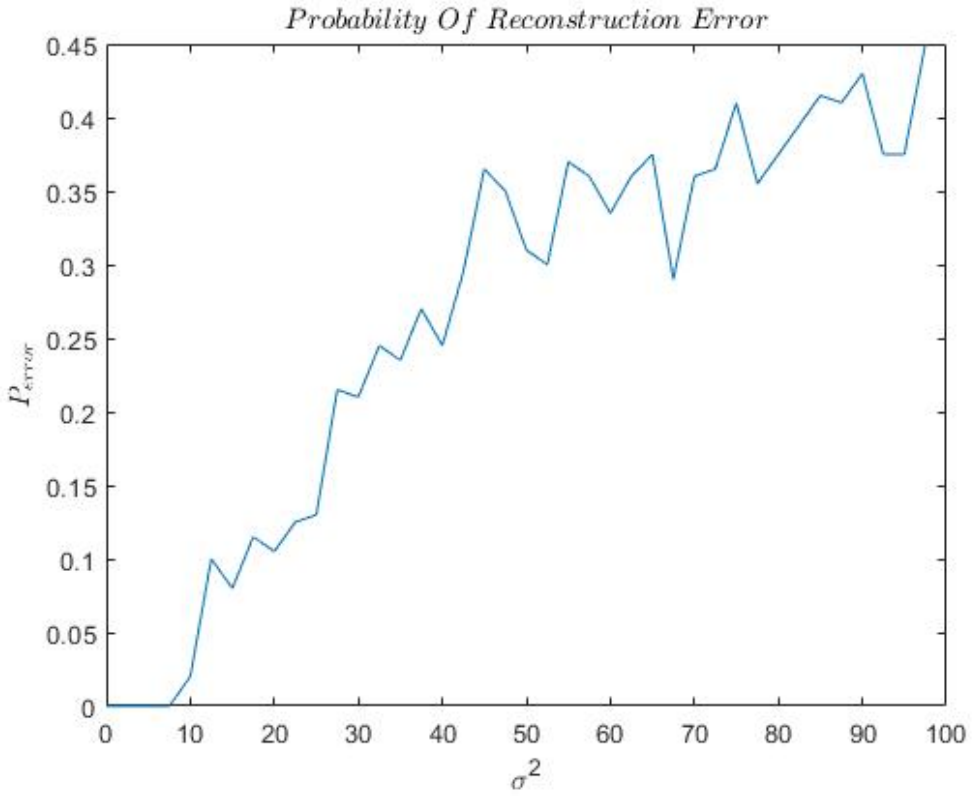
خروجی تابع Combine



خطای بازسازی

ج:

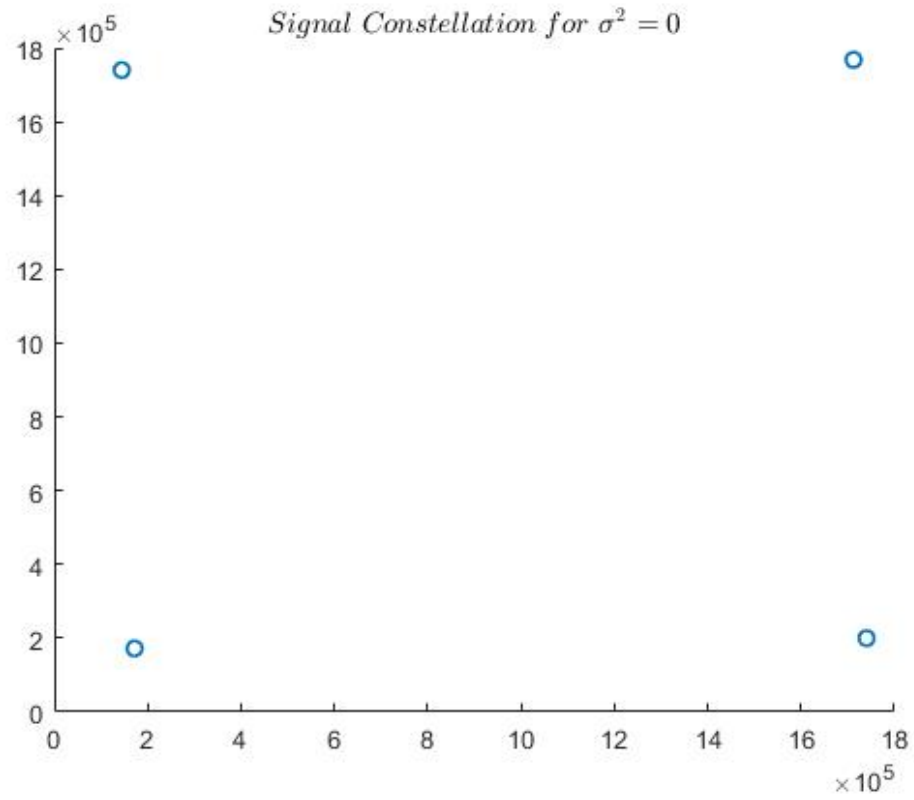
با استفاده از دستور normrnd در متلب که متغیر رندوم با توزیع گوسی و میانگین و واریانس دلخواه ما تولید میکند، به سیگنال خروجی از تابع Channel نویز می افزاییم. نتیجه خواسته شده به شرح زیر میباشد:



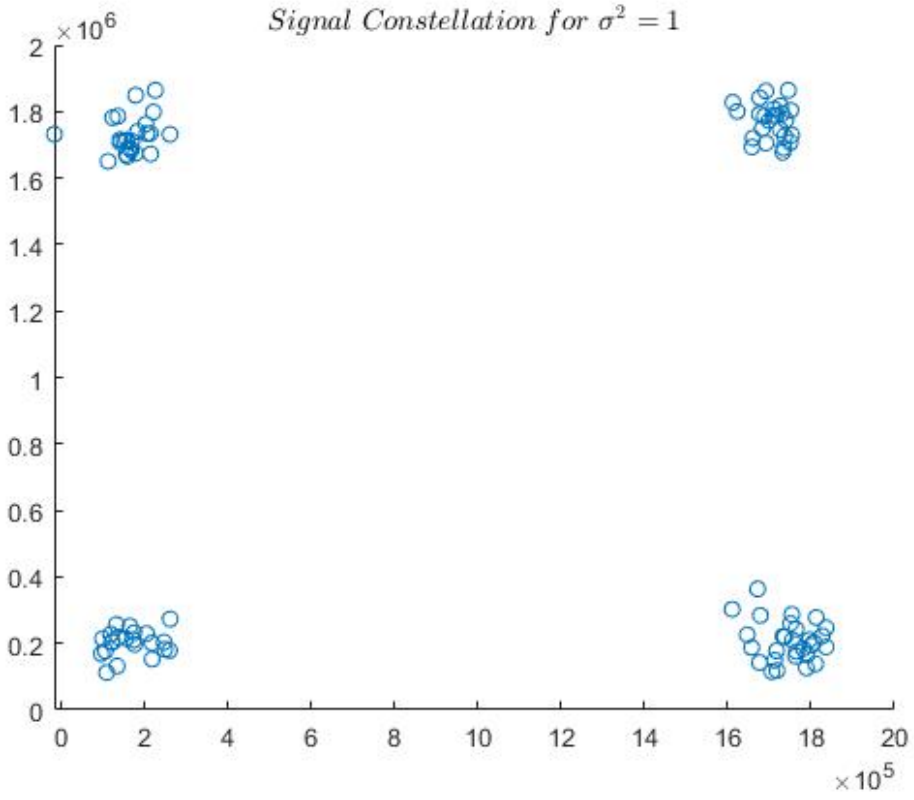
احتمال خطای بازسازی برحسب واریانس نویز کانال

واضح است انتظار داشتیم نتیجه ای مشابه با قسمت قبل را شاهد باشیم.البته تفاوت های ریزی نیز وجود دارد که در شکل واضح است. اما نکته مهم این است که همچنان مجموعه پروژه ما میتواند تا حدی اثر نویز را خنثی کند و بدون هیچ اشتباهی سیگنال را بازیابی کند. اما رفته رفته احتمال خطا بیشتر میشود تا اینکه اصلا به نزدیکی خطای ۵۰ درصد میرسیم که یعنی اصلا یک سیگنال رندوم جدید تولید شده است (مشابه با علتی که در قسمت قبل توضیح دادیم)
لازم به ذکر است رفتار کمی غیرخطی شکل فوق به این خاطر است که نمودار با ۴۰ نمونه ترسیم شده تا زمان زیادی برای پردازش صرف نشود.

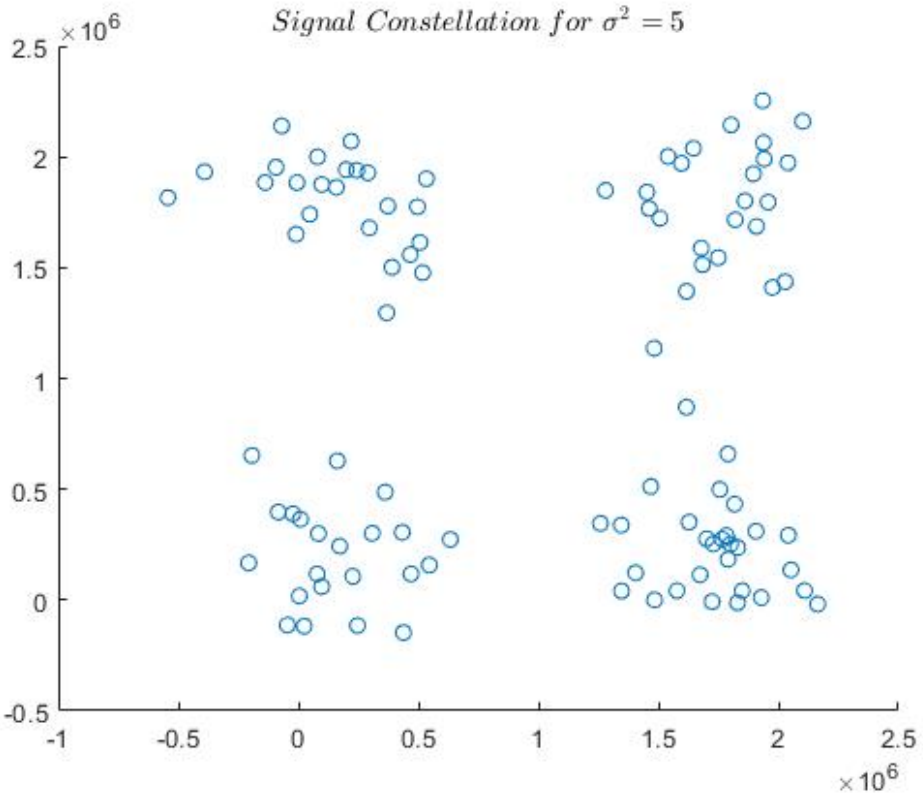
د:
بدون تکرار توضیحات مشابه قبل، نتایج شبیه سازی را با هم میبینیم:



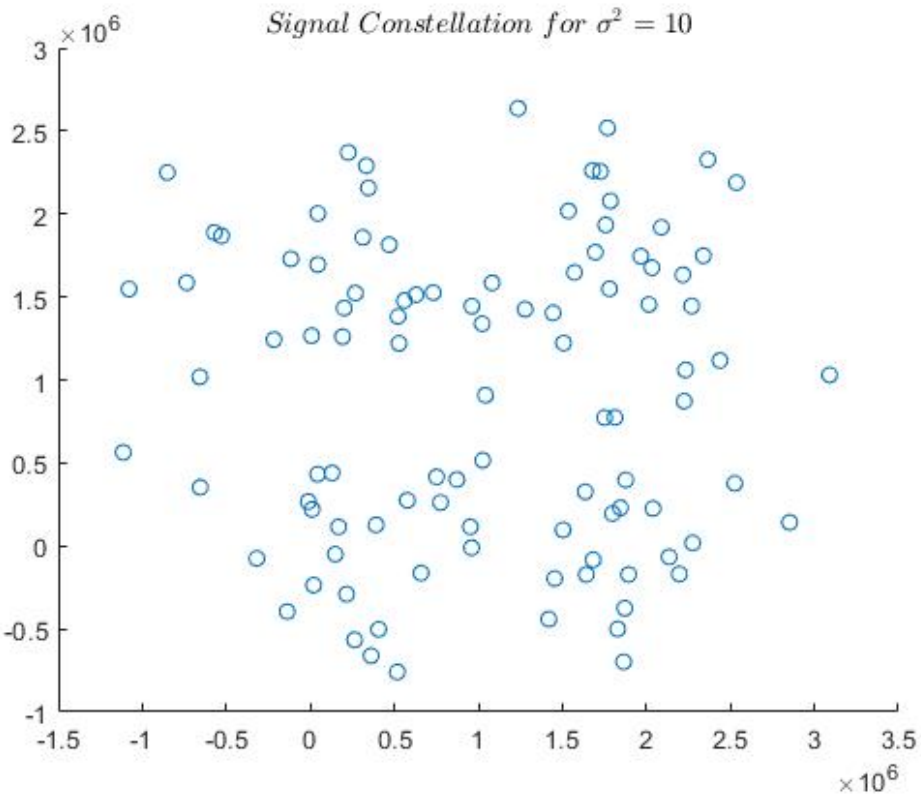
نویز با واریانس $\sigma^2 = 0$



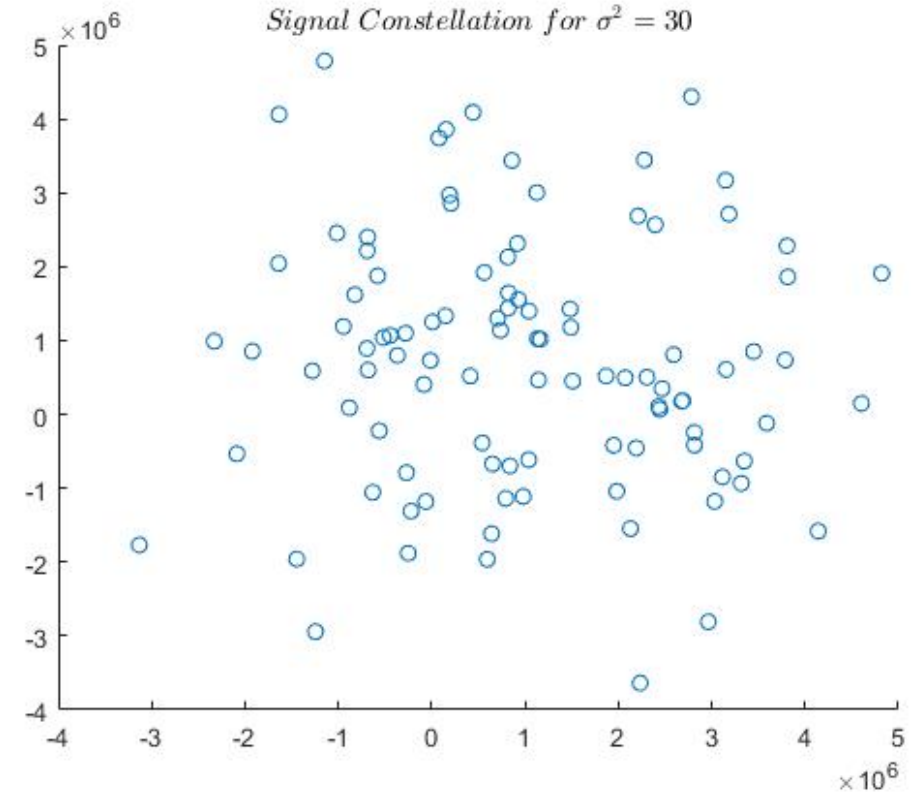
نویز با واریانس $\sigma^2 = ۱$



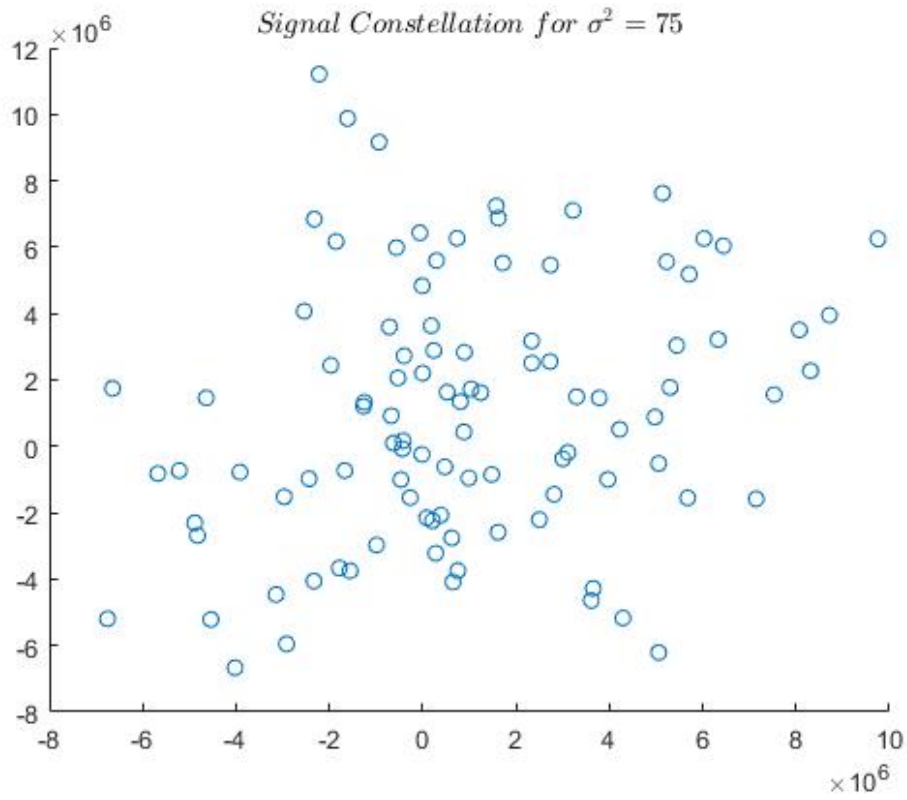
نویز با واریانس $\sigma^2 = ۵$



نویز با واریانس $\sigma^2 = ۱۰$



نویز با واریانس $\sigma^2 = ۳۰$



نویز با واریانس $\sigma^2 = ۷۵$

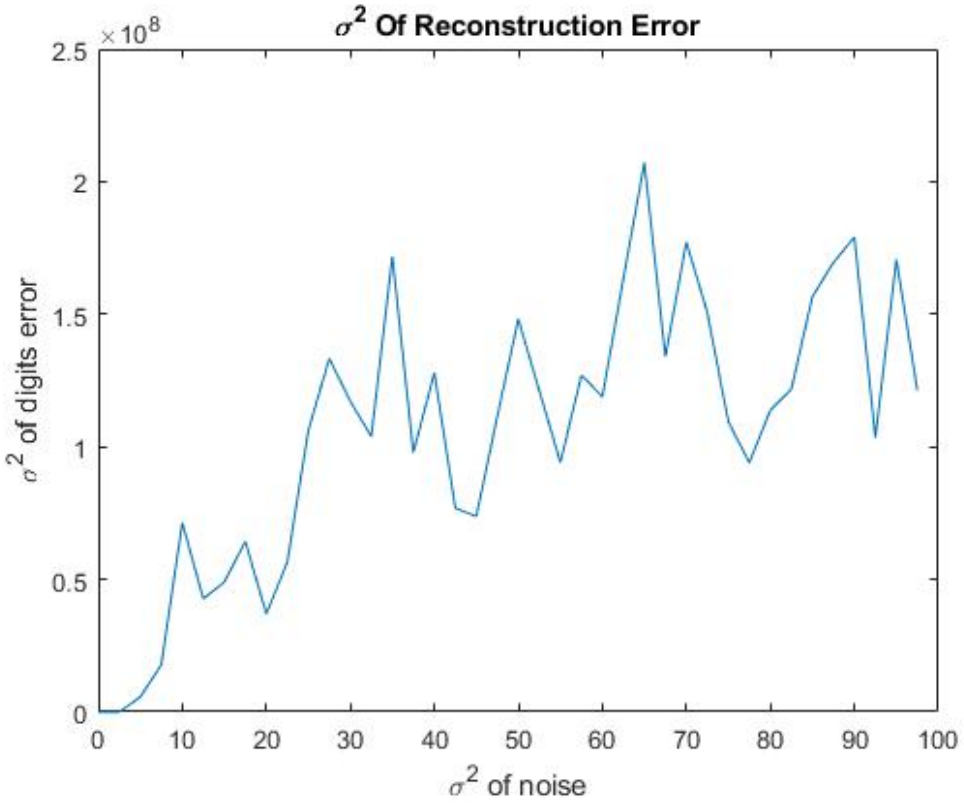
میبینیم که مدولاسیون FSK از خود مقاومت خوبی نشان داده و منظومه های سیگنال شکل های خوب و متراکمی هستند. البته که بدیهتا در این حالت نیز به ازای واریانس نویز بزرگ، منظومه از تراکم خارج شده و در کل صفحه پخش میشود.

۴ انتقال دنباله ای از اعداد ۸ بیتی

در این بخش ابتدا سیگنال هایی با اندازه های بین مقادیر ۰ تا ۲۵۵ را جداگانه به باینری تبدیل میکنیم و سپس آن هارا از طریق سازوکاری که تولید کرده ایم مدوله میکنیم. مجددا بیت های باینری مدوله شده را به اعداد دسیمال تبدیل میکنیم و انتظار داریم اعدادی که تولید کرده ایم با چیزی که ارسال کرده ایم مشابه باشند.

۱.۴: تابع SourceGenerator را مطابق عملکردی که باید طراحی کرده ایم. لازم به ذکر است که تابع bi۲de استفاده شده در این قسمت، یک ماتریس خروجی میدهد که ما سطر های مختلف این ماتریس را در کنار هم قرار داده ایم تا خروجی دلخواه تولید شود.
برای تابع SourceDecoder نیز روند مشابهی طی کردیم و از تابع de۲bi استفاده کرده و ۸ بیت را با استفاده از این تابع به اعداد دسیمال تبدیل کرده ایم.

۲.۴: طبق تعریف سوال، خطای بازسازی برابر با مجذور اختلاف اعداد است. با این تعریف نتیجه این بخش مشابه زیر میباشد:

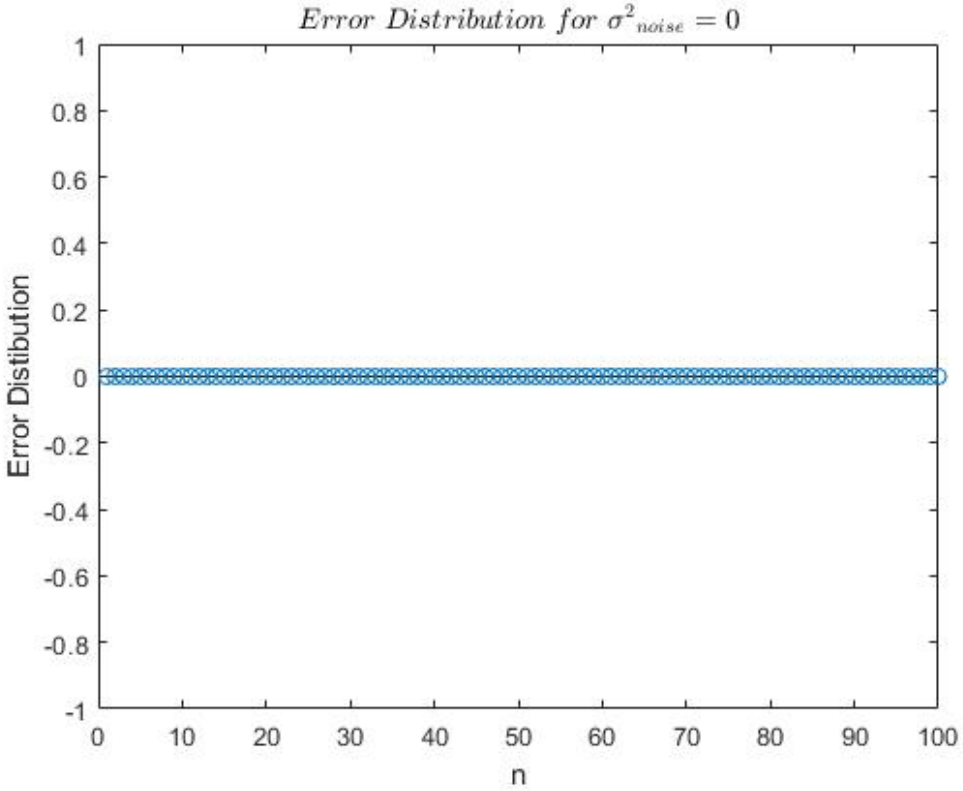


واریانس خطا برحسب واریانس نویز

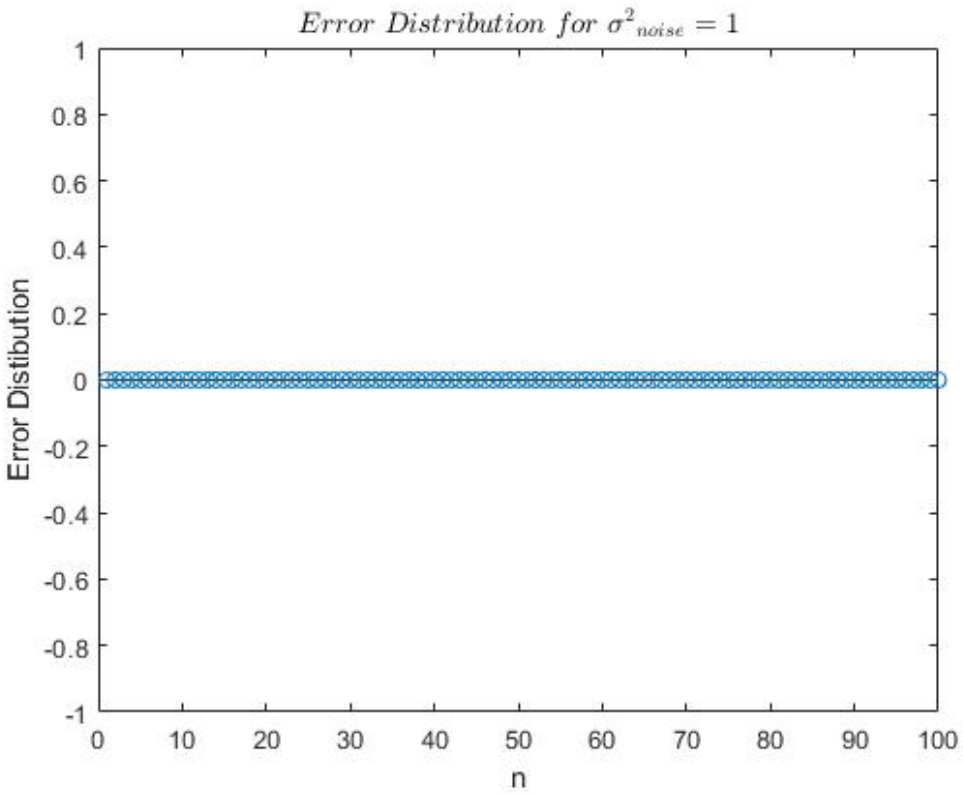
واضح است که زمانی که واریانس نویز کم است، بازسازی اعداد دسیمال بدون خطا بوده و بنابراین واریانس خطا نیز صفر است. رفته رفته با افزایش واریانس نویز، تعداد و مقدار خطای بازسازی اعداد دسیمال بیشتر شده به نحوی که به ازای نویز های بسیار بزرگ، اعداد دسیمال های بازسازی شده را میتوان اعداد رندوم بین ۰ تا ۲۵۵ فرض کرد.
برای همین واریانس خطا نیز یک متغیر تصادفی جدید است که میتواند مقادیر بسیار بزرگی اتخاذ کند که در تصویر

نیز واضح است.

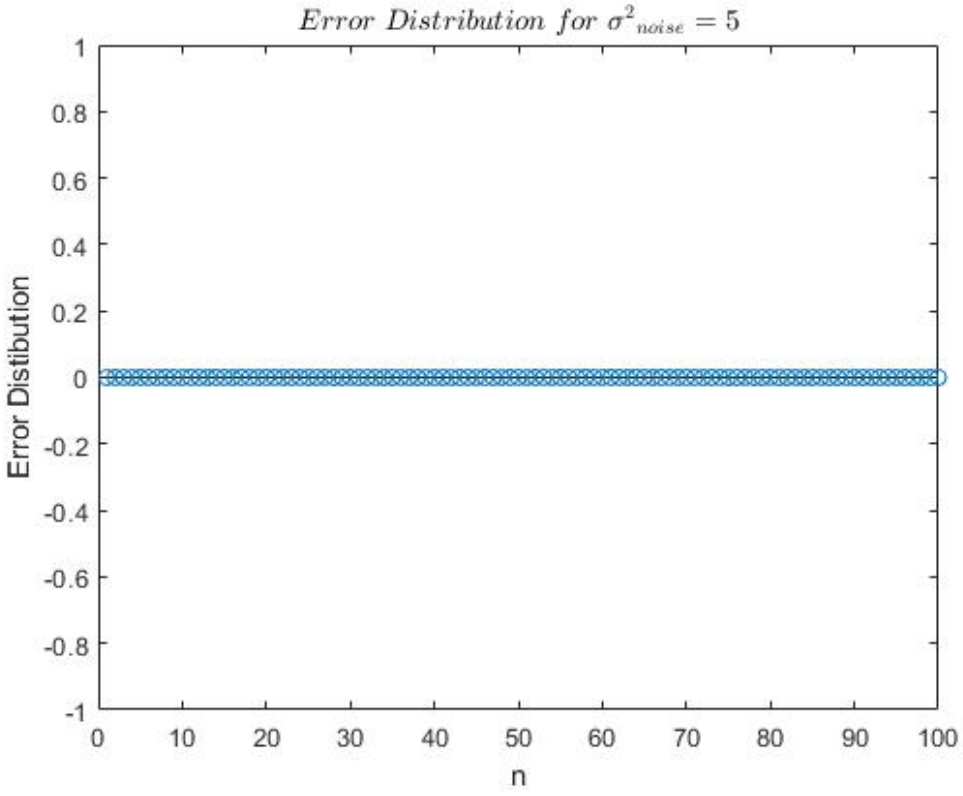
۳.۴: خواسته مساله را به ازای واریانس های مختلف ترسیم میکنیم که نتیجه به شرح زیر است:



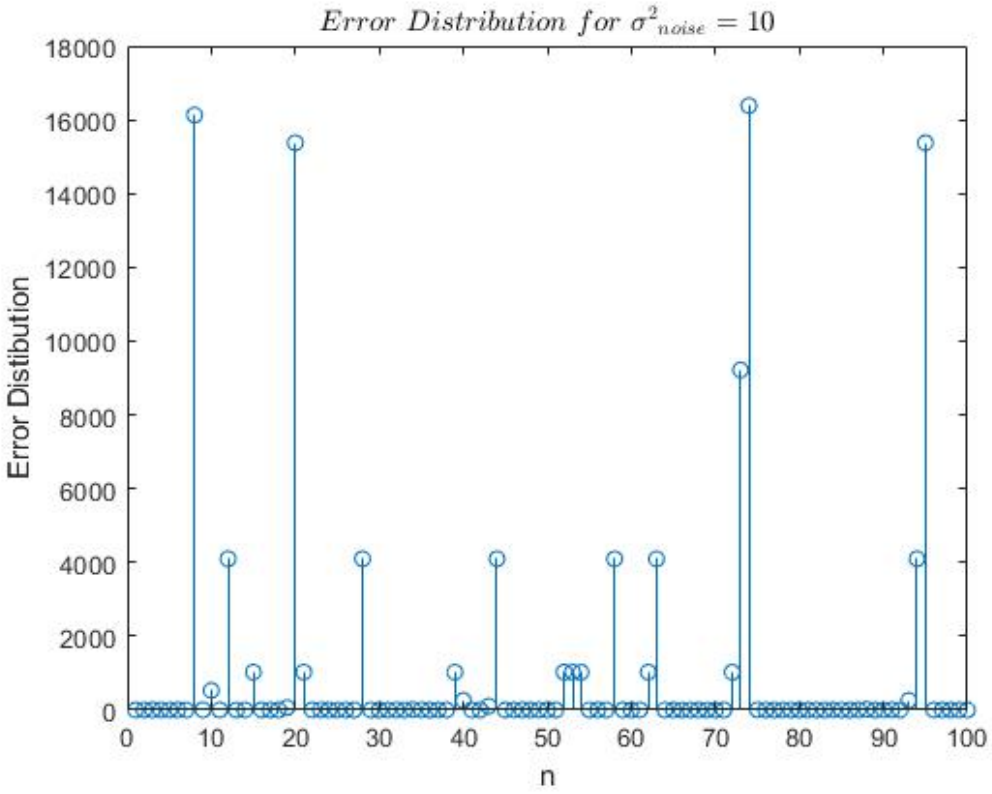
تابع توزیع خطا برای $\sigma^2_{noise} = 0$



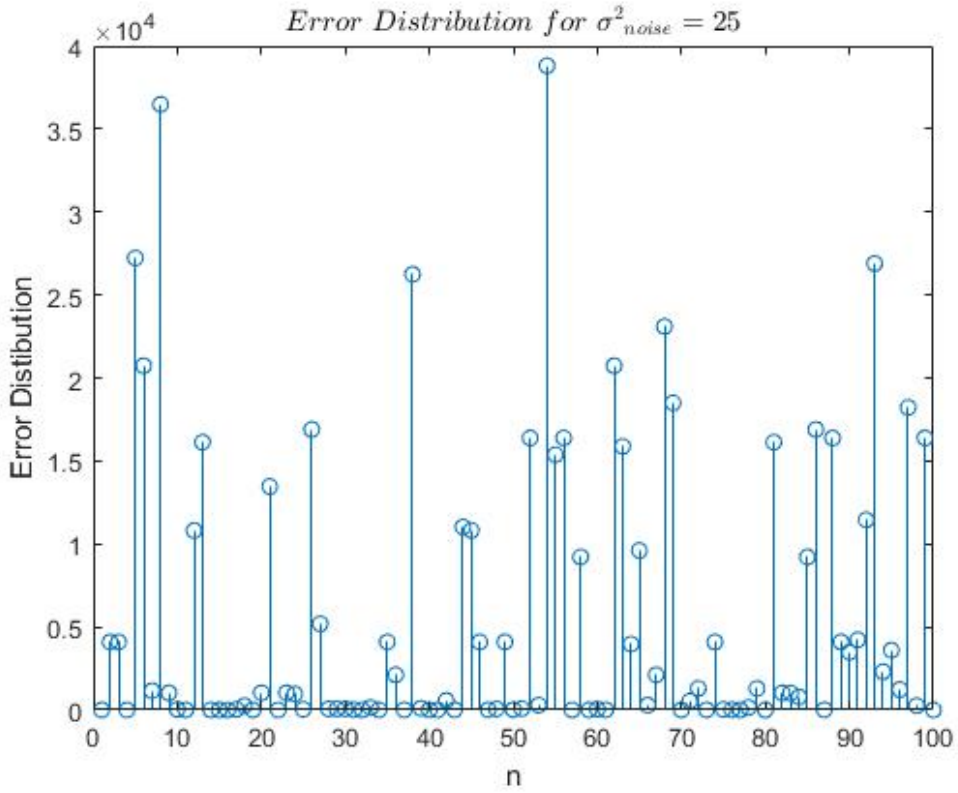
تابع توزیع خطا برای ۱ = σ^2_{noise}



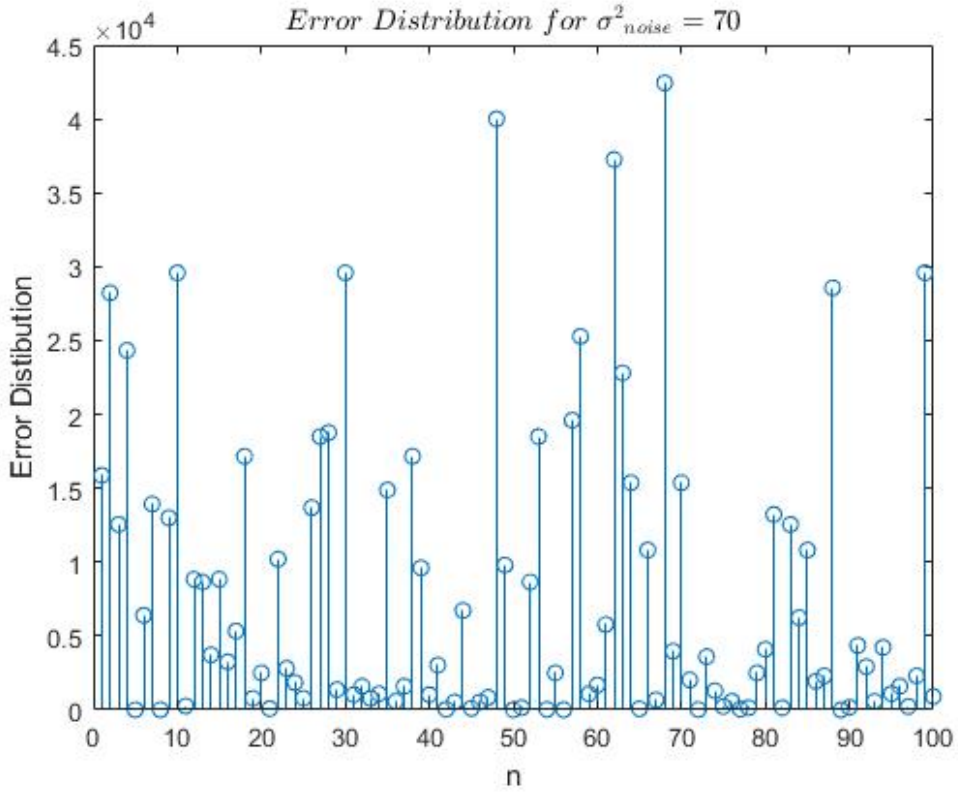
تابع توزیع خطا برای ۵ = $\sigma^{۲}_{noise}$



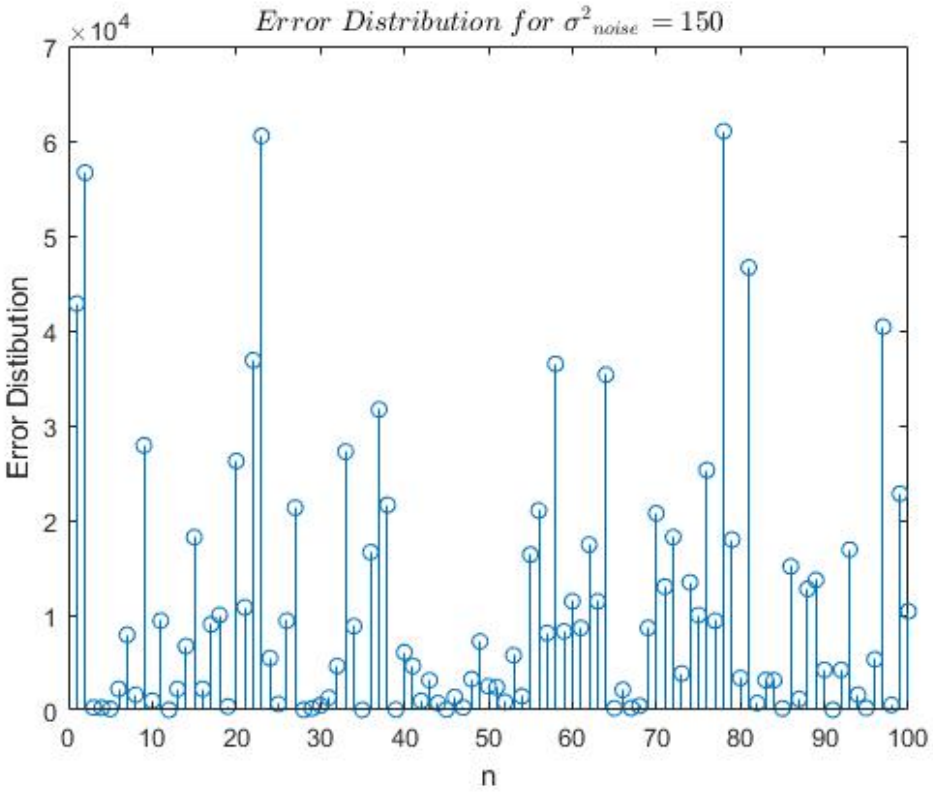
تابع توزیع خطا برای ۱۰ = $\sigma^{۲}_{noise}$



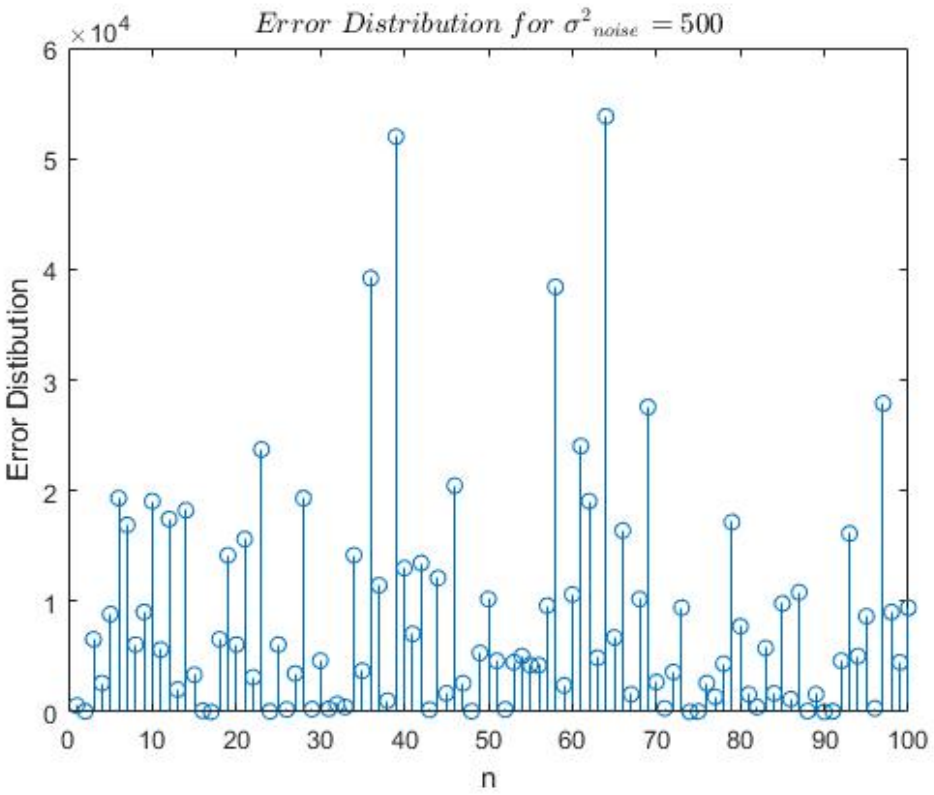
تابع توزیع خطا برای $\sigma^2_{noise} = ۲۵$



تابع توزیع خطا برای $\sigma^2_{noise} = ۷۰$



تابع توزیع خطا برای $\sigma^2_{noise} = ۱۵۰$



تابع توزیع خطا برای $\sigma^2_{noise} = ۵۰۰$

مطابق انتظار، توزیع خطا برای واریانس های کم توزیع ثابت صفر است. چرا که بازسازی اعداد دسیمال بدون هیچ خطایی انجام میشود. اما با افزایش نویز خطا کم کم در باز سازی ظاهر میشود تا جایی که برای نویز های بزرگ، عملا توزیع خطا را میتوان حاصل تفریق دو عدد رندوم میان ۰ تا ۲۵۰ دانست برای همین خود تابع توزیع برای واریانس های بزرگ نیز میتواند تمام مقادیر این بازه را (البته مجذور آن را) اختیار کند. پس انتظار داریم هر عددی در بازه $۰^۲$ الی $۲۵۵^۲$ را ملاحظه کنیم که همینطور نیز هست.

۴.۴:

توزیع خطا را هنگامی که نویز به بینهایت میل میکند میابیم. لازم به ذکر است که گفتیم که زمانی که نویز به بینهایت میل میکند سیگنال بازسازی شده کاملا یک سیگنال رندوم جدید است و کلید حل این مسئله، در مستقل بودن سیگنال بازسازی شده از سیگنال ارسالی است. خواهیم داشت:

$$X_n(x) \sim Uinform(۰, ۲۵۵)\ and\ Y_n(x) \sim Uniform(۰, ۲۵۵)\ and\ independent \rightarrow$$

$$Z = X - Y \sim \Lambda(-۲۵۵, ۲۵۵)$$

پس با توجه به توزیع خطا، واضح است که واریانس با رفتاری که از شبیه سازی دیدیم متشابه است.

۵ کدینگ منبع

۱.۵: با فرض منبع داده شده و کدینگ های مفروض:

آ:
واضح است کد مربوط به دو حرف b و d یکسان و برابر ۱۰ میباشد. بنابراین گیرنده با مشاهده کد ۱۰ نمیتواند تشخیص دهد کدام یک از حروف b یا d ارسال شده بوده است

ب:
کدگذاری سمبل ها نباید به نحوی باشد که کنارهم قرار گرفتن آن ها باعث تولید یک کد دیگر از همان مجموعه شود. یعنی فرض کنید که فرستنده 'ab' را ارسال میکند که به ۰۱۰ کد میشود. همینطور اگر 'd' را کد کرده و ارسال کند لازم است ۰۱۰ ارسال شود. حال گیرنده چگونه میتواند تشخیص دهد تفاوت میان این دو کدینگ یکسان ۰۱۰ چیست و در نتیجه این دیکود را نمیتوان تشخیص داد

ج:
همانطور که مساله نیز توضیح داده است، با این نوع کدینگ نمیتوان تصمیم آنی گرفت. یعنی مثلا فرض کنید در گیرنده بیت ۰ دیده میشود. حال میتواند تصمیم بگیرد که این ۰ را به حرف a دیکود کند و میتواند صبر کند تا بیت بعدی را دریافت کند. حال فرض کنید بیت بعدی ۱ بود. مجددا میتواند این مجموعه ۰۱ را به حرف b دیکود کند یا میتواند منتظر بماند تا بیت بعدی دریافت شود و همین مشکل تا آخر ادامه میابد.

۲.۵:
با توجه به راهنمایی خود مساله، موقتا مساله لاگرانژ را برای حالت تساوی حل میکنیم و از نتیجه آن استنباط لازم را انجام میدهیم:

دو معادله زیر را تشکیل میدهیم:

$$۱ : \sum_{i=۱}^{M} ۲^{-l_i} = \frac{۱}{۲^{l_a}} + \frac{۱}{۲^{l_b}} + \frac{۱}{۲^{l_c}} + \frac{۱}{۲^{l_d}} + \frac{۱}{۲^{l_e}} + \frac{۱}{۲^{l_f}} = ۱$$

$$\nabla F(l_a, l_b, l_c, l_d, l_e, l_f) = \nabla \sum_{i=۱}^{M} p_i l_i = \lambda \nabla \sum_{i=۱}^{M} ۲^{-l_i} = \lambda \nabla G(l_a, l_b, l_c, l_d, l_e, l_f) \rightarrow$$

$$۲ : < F_{l_a}, F_{l_b}, F_{l_c}, F_{l_d}, F_{l_e}, F_{l_f} > = < \lambda G_{l_a}, \lambda G_{l_b}, \lambda G_{l_c}, \lambda G_{l_d}, \lambda G_{l_e}, \lambda G_{l_f} >$$

از حل دو معادله فوق نتیجه زیر حاصل میشود:

$$l_a = ۱, l_b = ۲, l_c = ۳, l_d = ۴, l_e = ۵, l_f = ۵$$

۳.۵:
با توجه به حالاتی که در قسمت قبل توضیح داده شده و نتیجه ای که خودمان پیدا کردیم، یک کدگذاری ممکن به صورت زیر خواهد بود:

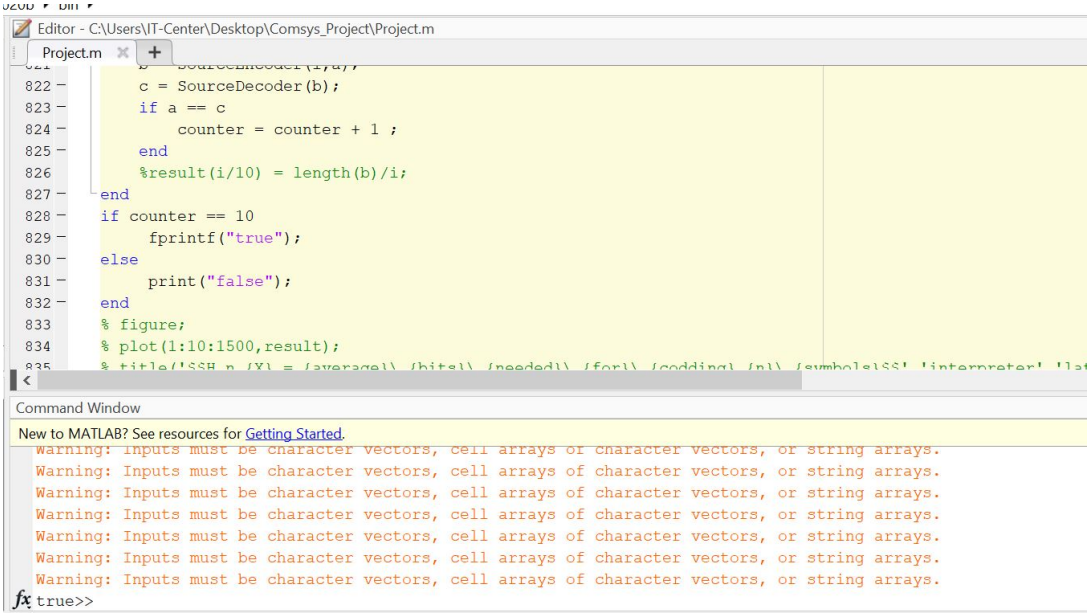
$$a = ۰, b = ۱۰, c = ۱۱۰, d = ۱۱۱۰, e = ۱۱۱۱۰, f = ۱۱۱۱۱$$

:۴.۵
طول متوسط کد ها برابر است با:

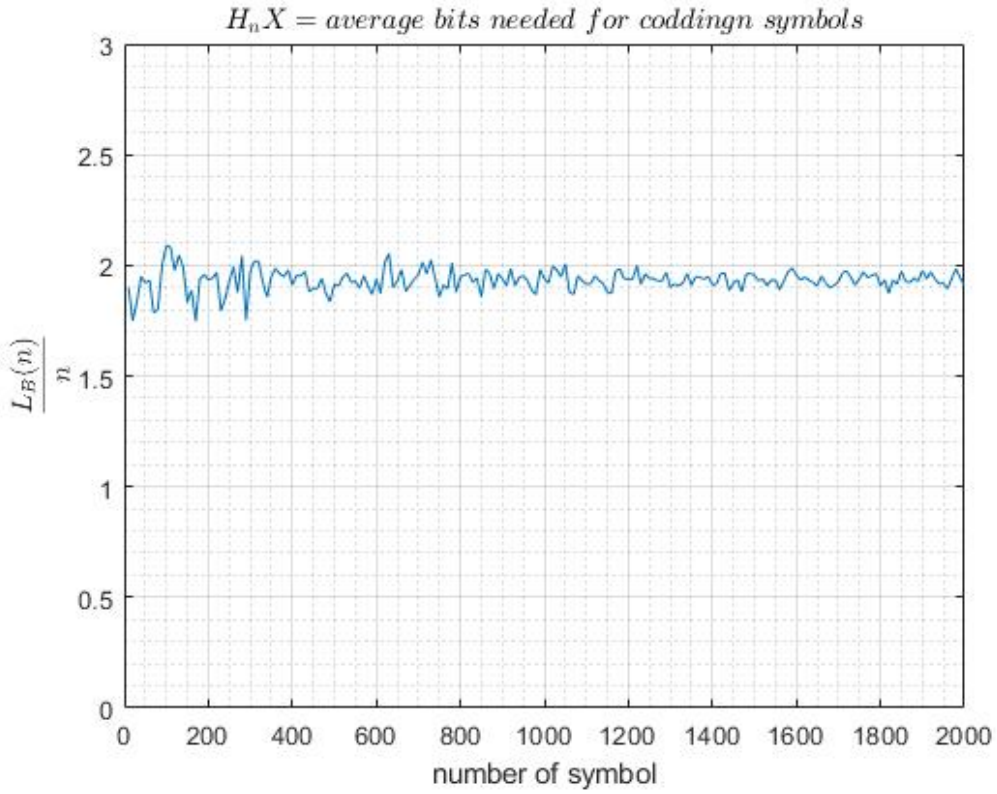
$$\bar{l} = \frac{۱ + ۲ + ۳ + ۴ + ۵ + ۵}{۶} = ۳/۳۳$$

:۵.۵ الی ۷.۵
توابع خواسته شده در این بخش هارا تولید کردیم. نکته خاصی در تولید این توابع وجود نداشت و توضیحات لازمه را در صورت نیاز در کد کامنت کرده ایم. احتمالات موجود در قسمت اول سوال که بیان شد(حروف) را با اعداد در اینجا هم ارز کرده ایم و احتمالات هر کدام را با توجه به آن نوشته و به طور کامل توابع در کد نشان داده شده اند

:۸.۵
برای قطعه کد این بخش نیز در هر مرحله تصویر را بررسی کنیم و اگر دو حالت برابر بودند راcounter یک واحد اضافه میکنیم و اگر در for ما برابرcounter ۱۰شود این یعنی نتیجه درستی گرفته ایم و true پرینت می شود



:۹.۵



$H_n(X) = \frac{L_avg}{n}$ بر حسب n

واضح است که برای مقادیر n کوچک مقدار مورد بررسی عدد مشخصی ندارد. اما با افزایش تعداد سیمبول ها، مقدار میانگین طول به عددی نزدیک ۲ میل میکند. حالا نتیجه ریاضی را بررسی میکنیم:

$$l_{\cdot} = ۱, p_{\cdot} = \frac{۱}{۲}; l_{(۲۵۰)} = ۲, p_{(۲۵۰)} = \frac{۱}{۴}; l_{(۱۰۰)} = ۳, p_{(۱۰۰)} = \frac{۱}{۸}; l_{(۱۵۰)} = ۴, p_{(۱۵۰)} = \frac{۱}{۱۶}$$

$$l_{(۲۰۰)} = ۵, p_{(۲۰۰)} = \frac{۱}{۳۲}; l_{(۵۰)} = ۵, p_{(۵۰)} = \frac{۱}{۳۲}$$

$$\bar{length} = \frac{۱}{۲} \times ۱ + \frac{۱}{۴} \times ۲ + \frac{۱}{۸} \times ۳ + \frac{۱}{۱۶} \times ۴ + \frac{۱}{۳۲} \times ۳ + \frac{۱}{۳۲} \times ۵ = ۱/۹۳۷۵$$

پس در حالت تئوریک انتظار داریم به عدد ۱.۹۳۷۵ میل کنیم که در تصویر حاصل از شبیه سازی نیز این کاملا مشهود است.