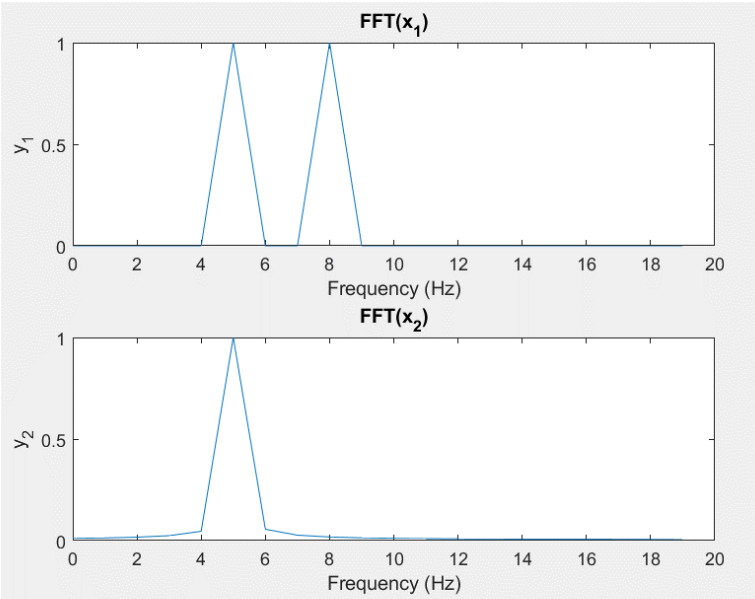
بخش اول

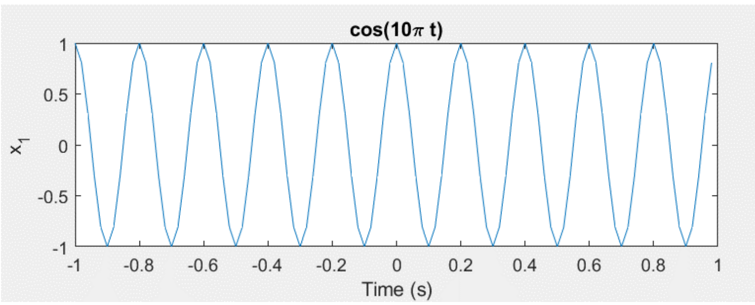
0. توجیه رزولوشن فرکانسی

همانطور که در دستور کار گفته شده است، در حالتی که فرکانس‌های تابع exp را برابر با 5 و 8 در نظر بگیریم، دو قله بر روی 5 و 8 به وضوح قابل مشاهده هستند. اما اگر این مقادیر را 5 و 5.1 در نظر بگیریم، چون تفاوت آن‌ها کمتر از مقدار رزولوشن فرکانس (1 هرتز) است، فقط یک قله با کمی نویز قابل مشاهده است. این مورد در تصویر زیر آورده شده است:

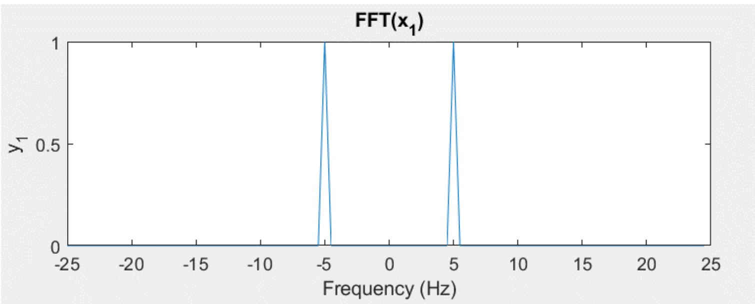


1. تبدیل فوریه سیگنال $cos(10\pi t)$

الف) نمودار سیگنال



ب) نمودار اندازه تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

می‌دانیم تبدیل فوریه تابع $cos(\omega_0 t)$ به صورت زیر محاسبه می‌شود.

$\mathcal{F}\{\cos(\omega_0 t)\} = \pi\delta(\omega - \omega_0) + \pi\delta(\omega + \omega_0)$

از طرفی با توجه به اینکه در متلب تبدیل فوریه را normalize می‌کنیم، ضرایب π را از پاسخ حذف می‌کنیم. با جایگذاری مقادیر، نتیجه به صورت زیر خواهد بود:

$\omega_0 = 10\pi \longrightarrow \mathcal{F}_{\mathcal{N}}\{cos(10\pi t)\} = \delta(\omega - 10\pi) + \delta(\omega + 10\pi)$

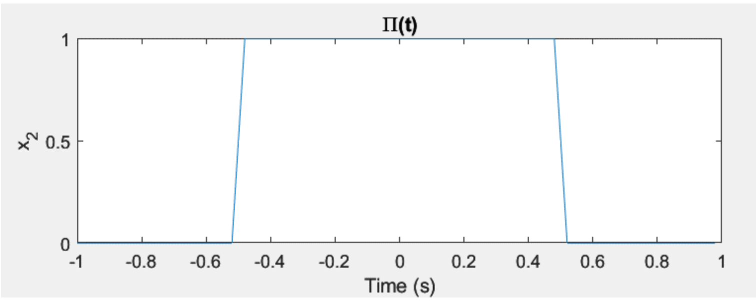
از طرفی نمودارها به جای اینکه بر اساس ω رسم شده باشند، بر حسب f رسم شده‌اند. در نتیجه باید این تغییر متغیر را نیز لحاظ کنیم:

$\omega = 2\pi f \longrightarrow \mathcal{F}_{\mathcal{N}}\{cos(10\pi t)\} = \delta(f - 5) + \delta(f + 5)$

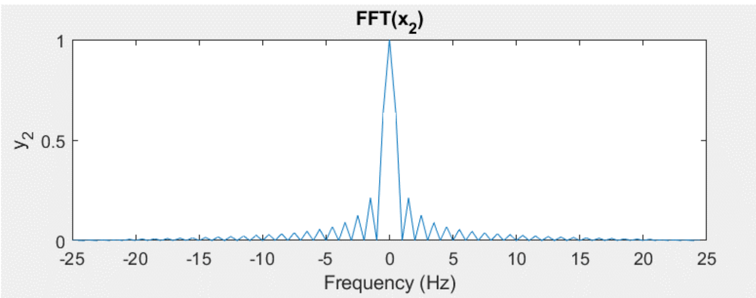
همانطور که مشاهده می‌شود، محاسبات تئوری با مقدار بدست آمده مطابقت دارد.

2. تبدیل فوریه سیگنال $\Pi(t)$

الف) نمودار سیگنال



ب) نمودار اندازه تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

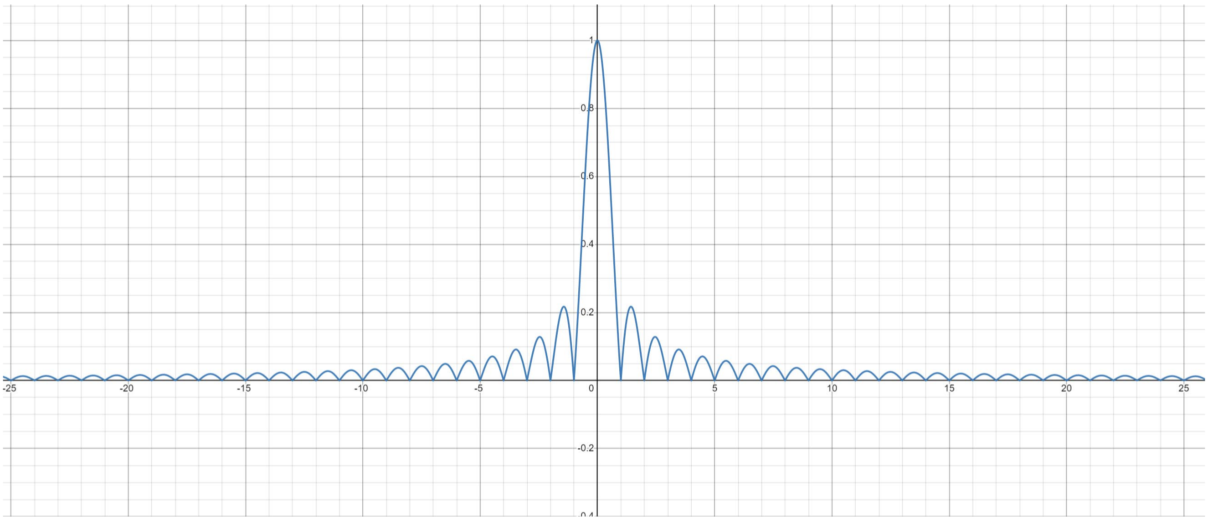
تبدیل فوریه تابع $\Pi(t)$ به صورت زیر محاسبه می‌شود:

$\mathcal{F}\{\Pi(t)\} = sinc_{\pi}\left(\frac{\omega}{2\pi}\right)$

از طرفی باید مبنا را از ω به f تغییر دهیم:

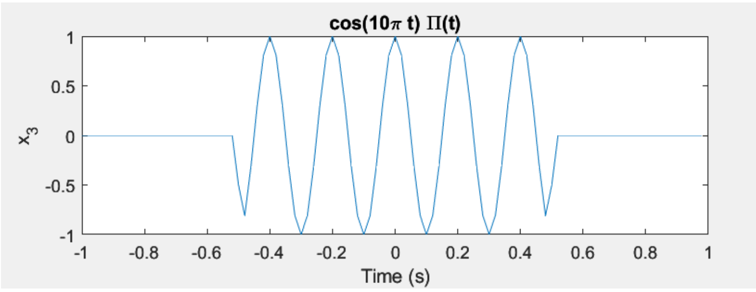
$\omega = 2\pi f \longrightarrow \mathcal{F}\{\Pi(t)\} = sinc_{\pi}(f)$

با توجه به اینکه اندازه تبدیل فوریه رسم شده است، تابع $|sinc_{\pi}(f)|$ مد نظر است. نمودار این تابع در desmos رسم شده و تصویر آن در ادامه آورده شده است. همانطور که مشاهده می‌شود، این نمودار با نمودار رسم شده در متلب مطابقت دارد.

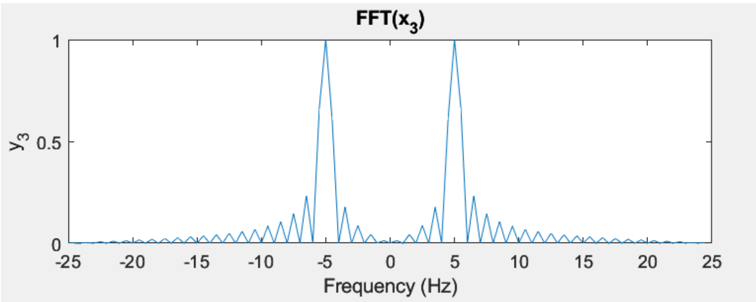


3. تبدیل فوریه سیگنال $cos(10\pi t)\,\Pi(t)$

الف) نمودار سیگنال



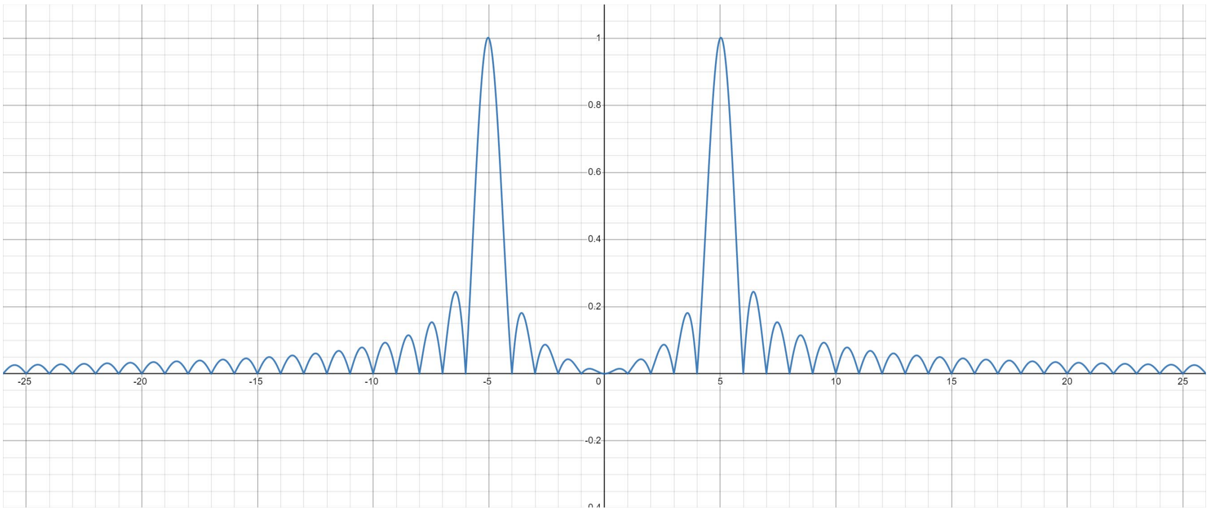
ب) نمودار اندازه تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

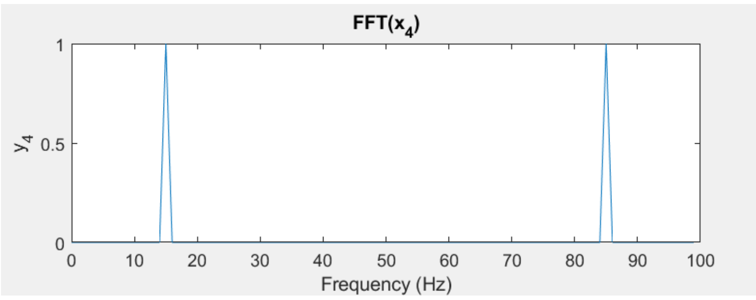
$$x_1 = cos(10\pi t)\,,\ x_2 = \Pi(t),\ x_3 = x_1 \times x_2 \longrightarrow \mathcal{F}\{x_3\} = \frac{1}{2\pi}(\mathcal{F}\{x_1\} * \mathcal{F}\{x_2\})$$
$$= \frac{1}{2\pi}\Big(\big(\delta(f-5)+\delta(f+5)\big) * sinc_\pi(f)\Big) = \frac{1}{2\pi}\big(\mathrm{sinc}_\pi(f-5)+\mathrm{sinc}_\pi(f+5)\big)$$
$$\longrightarrow |\mathcal{F}_{\mathcal{N}}\{x_3\}| = |sinc_\pi(f-5)+sinc_\pi(f+5)|$$

نمودار این تابع در desmos نیز رسم شده و در ادامه آورده شده است. همانطور که مشاهده می‌شود، مقدار تئوری بدست آمده با نمودار رسم شده در متلب مطابقت دارد.

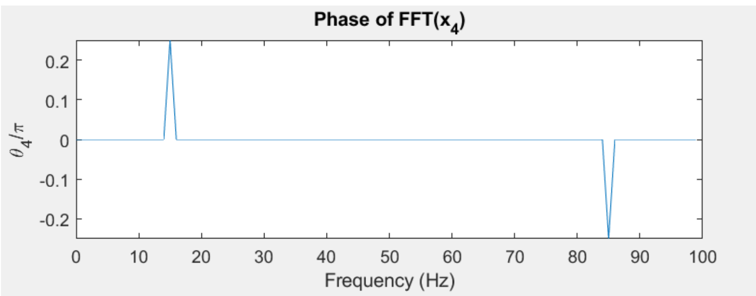


4. تبدیل فوریه سیگنال $cos\left(30\pi t+\frac{\pi}{4}\right)$

الف) نمودار اندازه تبدیل فوریه سیگنال



ب) نمودار فاز تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

ابتدا تبدیل فوریه تابع $cos(\omega_0 t+t_0)$ را محاسبه می‌کنیم:

$$\mathcal{F}\{cos(\omega_0 t+t_0)\}=\int_{-\infty}^{+\infty}\frac{e^{j(\omega_0 t+t_0)}+e^{-j(\omega_0 t+t_0)}}{2}e^{-j\omega t}\,dt=\pi e^{-jt_0}\delta(\omega+\omega_0)+\pi e^{jt_0}\delta(\omega-\omega_0)$$

حال تبدیل فوریه تابع $cos\left(30\pi t+\frac{\pi}{4}\right)$ را به صورت زیر بدست می‌آوریم:

$$\mathcal{F}\left\{cos\left(30\pi t+\frac{\pi}{4}\right)\right\}=\pi e^{-j\frac{\pi}{4}}\delta(\omega+30\pi)+\pi e^{j\frac{\pi}{4}}\delta(\omega-30\pi)$$

حال تغییر متغیر $\omega=2\pi f$ را انجام می‌دهیم و با توجه به اینکه باید اندازه تابع را نرمالایز کنیم، ضریب π را در نظر نمی‌گیریم:

$$\mathcal{F}_{\mathcal{N}}\left\{cos\left(30\pi t+\frac{\pi}{4}\right)\right\}=e^{-j\frac{\pi}{4}}\delta(f+15)+e^{j\frac{\pi}{4}}\delta(f-15)$$

با توجه به اینکه بازه را قرینه در نظر نگرفتیم و فرکانس را نیز برابر با 100 هرتز در نظر گرفتیم، پاسخ بالا به پاسخ زیر تبدیل می‌شود:

$$\mathcal{F}_{\mathcal{N}}\{cos\left(30\pi t+\frac{\pi}{4}\right)\} = e^{-j\frac{\pi}{4}}\delta(f-85)+e^{j\frac{\pi}{4}}\delta(f-15)$$

با توجه به اینکه تابع e^{jx} اندازه‌ای برابر با 1 دارد، اندازه تبدیل فوریه برابر با مقدار تابع ضربه در نقطه ضربه است. به همین دلیل است که پس از نرمال‌سازی، در نمودار اندازه تبدیل فوریه یک ضربه در نقطه 15 و یک ضربه در نقطه 85 وجود دارد.

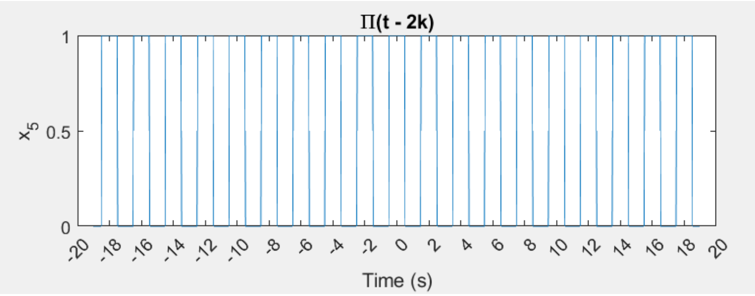
فاز تابع e^{jx} نیز برابر با x است و به همین دلیل فاز تبدیل فوریه از رابطه زیر بدست می‌آید:

$$\angle\left(\mathcal{F}\{cos\left(30\pi t+\frac{\pi}{4}\right)\}\right) = \frac{-\pi}{4}\delta(f-85)+\frac{\pi}{4}\delta(f-15)$$

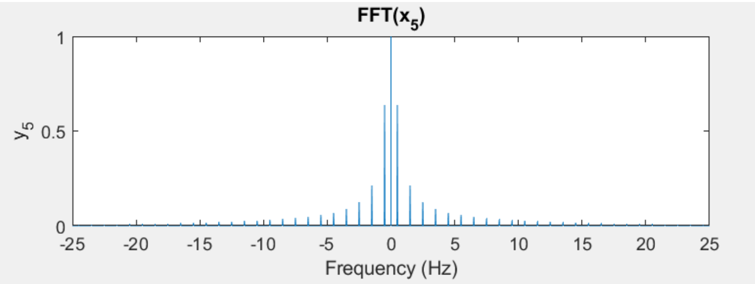
مقدار فاز نیز در متلب تقسیم بر π شده و به همین دلیل در نقطه 15 یک ضربه با اندازه $\frac{1}{4}$ و در نقطه 85 یک ضربه با اندازه $\frac{-1}{4}$ داریم.

5. تبدیل فوریه سیگنال $\sum_{k=-9}^{9}\Pi(t-2k)$

الف) نمودار سیگنال



ب) نمودار اندازه تبدیل فوریه سیگنال



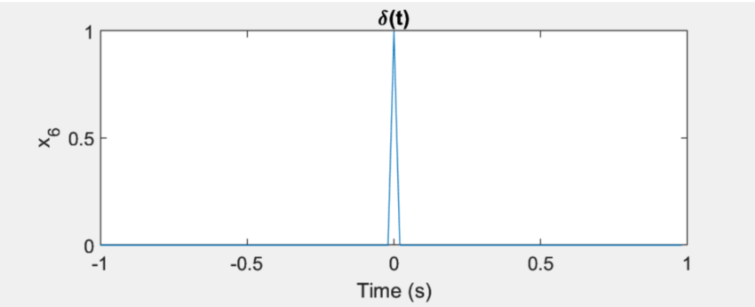
ج) تبدیل فوریه سیگنال‌های متناوب

پیش از تبدیل فوریه، با استفاده از سری فوریه توانستیم سیگنال‌های متناوب را در حوزه فوریه نشان دهیم. در واقع در سری فوریه از تعدادی سیگنال ویژه به فرم $e^{j\frac{2\pi}{T}kt}$ استفاده کردیم که نمودار آن فرم گسسته داشت. دلیل استفاده از تبدیل فوریه این است که بتوانیم توابع غیرمتناوب را در حوزه فوریه نشان دهیم که در این صورت از تمامی سیگنال‌های ویژه به فرم $e^{S_k t}$ استفاده می‌کنیم. حال وقتی می‌توانیم سیگنال‌های متناوب را با فرم خاصی از این سیگنال‌های ویژه نشان دهیم، نیازی به بقیه سیگنال‌های ویژه نخواهیم داشت و در واقع در این توابع، تبدیل فوریه ضریبی از سری فوریه خواهد بود. با توجه به اینکه سری فوریه تابع $\Pi(t)$، تابع $sinc$ است، تبدیل فوریه آن نیز همین تابع خواهد بود. فواصل ضربه‌ها (دوره تناوب) نیز به ضریب k که در این تابع برابر با 2 است وابسته است. در نتیجه دوره تناوب این قطار برابر با 2 خواهد بود.

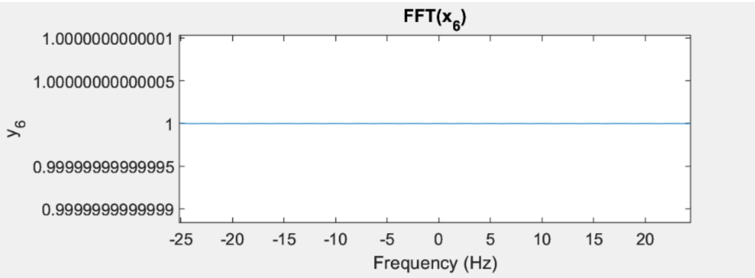
بخش دوم

1. تبدیل فوریه تابع $\delta(t)$

الف) نمودار سیگنال



ب) نمودار اندازه تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

شرح مشاهده: ناپیوستگی، شدیدترین تغییرات در حوزه زمان است و تابع دلتا نیز ناپیوستگی دارد. تبدیل فوریه ما را به حوزه فرکانس می‌برد که در آنجا تغییرات شدید زمان معادل فرکانس‌های بالاتر می‌شود.

ناپیوستگی شامل بزرگ‌ترین فرکانس‌ها یعنی بینهایت و منفی بینهایت می‌شود. از آنجا که تعداد محدودی فرکانس برای بیان کردن آن پاسخگو نیست، باید از منفی بینهایت تا بینهایت گسترده باشد.

این نکته در نمودار رسم شده نیز قابل رویت است. تبدیل فوریه تابع دلتا، تابع ثابت بوده و همه فرکانس‌ها را در بر می‌گیرد.

محاسبه تبدیل فوریه تابع دلتا:

$$\mathcal{F}\{\delta(t-t_0)\} = \int_{-\infty}^{\infty} \delta(t-t_0)e^{-j\omega t}dt = e^{-j\omega t_0}$$

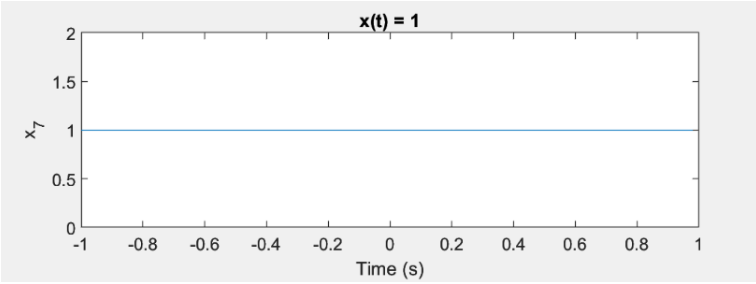
در اینجا $t_0 = 0$ است. پس داریم:

$$\mathcal{F}\{\delta(t)\} = e^{-j\omega 0} = 1$$

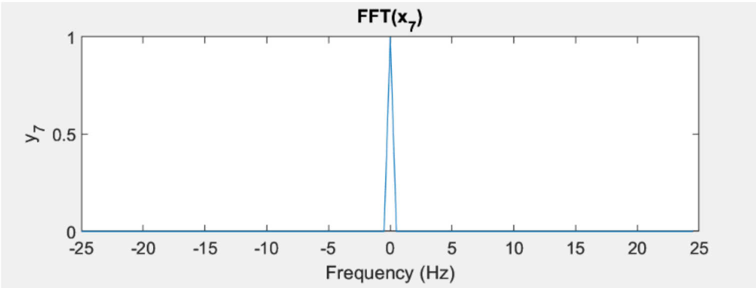
یعنی در هر نقطه 1 بوده و با در نظر گرفتن $\omega = 2\pi f$، یعنی در کل فرکانس‌ها 1 می‌باشد.

2. تبدیل فوریه تابع $x(t) = 1$

الف) نمودار سیگنال



ب) نمودار اندازه تبدیل فوریه سیگنال



ج) محاسبه تئوری تبدیل فوریه

شرح مشاهده: تابع ثابت هیچ تغییراتی در حوزه زمان ندارد و پس از تبدیل فوریه که به حوزه فرکانس می‌رویم، کمترین فرکانس‌ها را خواهیم داشت.

این یعنی تغییرات حوزه زمان با فرکانس‌های پایین قابل بیان بوده و با فقط یک ضربه توصیف شده است.

این نکته در نمودار رسم شده قابل رویت است. تبدیل فوریه تابع ثابت، یک ضربه شده که فقط شامل فرکانس‌های پایین است.

محاسبه تبدیل فوریه تابع ثابت $x(t) = c$:

$$\mathcal{F}\{c\} = \int_{-\infty}^{\infty} c \times e^{-j\omega t} dt = c \times \int_{-\infty}^{\infty} e^{-j\omega t} dt = c \times 2\pi\delta(\omega)$$

در اینجا $c = 1$ است. پس داریم:

$\mathcal{F}\{1\} = 2\pi\delta(\omega)$

با توجه به اینکه در متلب تبدیل فوریه را normalize می‌کنیم، ضریب 2π را از پاسخ حذف می‌کنیم:

$\mathcal{F}_{\mathcal{N}}\{1\} = \delta(\omega)$

در نمودار تبدیل فوریه، محور افقی بر حسب فرکانس (f) رسم شده است ولی تبدیل فوریه حساب شده بر حسب فرکانس زاویه‌ای (ω) می‌باشد. پس باید این تغییر متغیر را لحاظ کنیم:

$\omega = 2\pi f \longrightarrow \mathcal{F}_{\mathcal{N}}\{1\} = \delta(f)$

همانطور که مشاهده می‌شود، محاسبات تئوری با نمودار رسم شده مطابقت دارد.

بخش سوم

1. ساخت Mapset

Mapset خواسته شده به صورت زیر است:

	1	2	3	4	5	6	7	8	
1	a	b	c	d	e	f	g	h	i
2	00000	00001	00010	00011	00100	00101	00110	00111	01
3									
4									
5									
6									
7									
8									

این Mapset در ابتدای برنامه از فایل mapset.mat لود شده و در متغیر `mapset` قرار می‌گیرد.

2. تابع coding_amp

این تابع با شناسه زیر در فایل coding_amp.m قرار گرفته است:

```
function signal = coding_amp(bin_msg, bitrate)
```

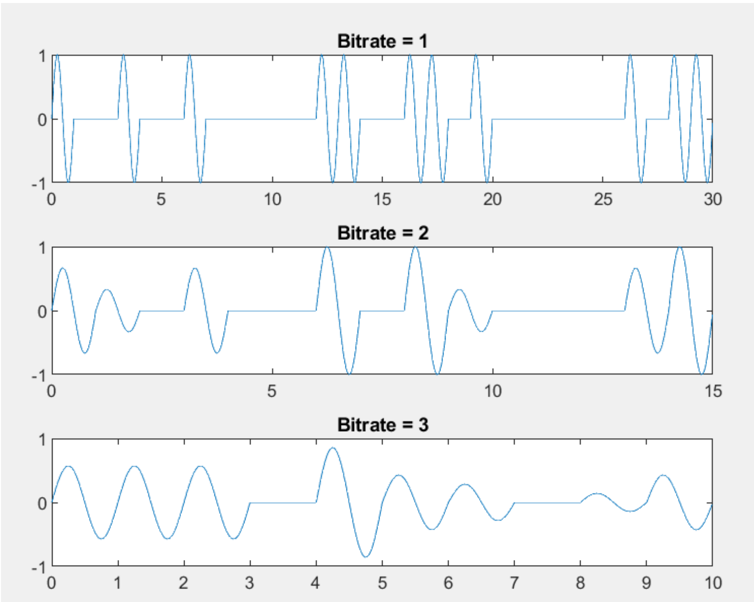
پیام ورودی به این تابع به صورت باینری است. برای تبدیل رشته به باینری از تابع `str2bin` موجود در فایل str2bin.m استفاده می‌شود.

تابع `coding_amp` برای استرینگ‌هایی که تعداد بیت‌های رشته باینری آن مضربی از bitrate نیست، تعدادی بیت 0 در انتهای راست رشته اضافه می‌کند تا طول رشته نهایی مضربی از bitrate شود. بیت‌های اضافه شده در بخش decoding حذف می‌شوند و رشته به صورت صحیح خوانده می‌شود.

خروجی این تابع یک سیگنال است که هر 100 سمپل آن معادل bitrate عدد بیت است.

3. خروجی تابع coding_amp

خروجی تابع به ازای سه مقدار 1، 2 و 3 برای bitrate به صورت زیر است. لازم به ذکر است که کلمه انتخاب شده برای encode کردن، کلمه `signal` است.



4. تابع decoding_amp

این تابع با شناسه زیر در فایل decoding_amp.m قرار گرفته است:

```
function binary = decoding_amp(signal, bitrate)
```

برای انجام correlation، انتگرال ضرب دو تابع $2sin(2\pi t)$ و 100 سمپل سیگنال حساب شده است (با استفاده از trapezoid).

به دلیل اینکه با وجود نویز دامنه تابع می‌تواند از محدودهٔ 1- تا 1 خارج شود، سیگنال‌های بیشتر از 1 به خود 1، و سیگنال‌های کمتر از 1- به 1- فیت شده‌اند.

سپس مقدار correlation در تعداد حالات دامنه برای بیت‌ریت مورد نظر ضرب شده که رند شده این حاصل، عدد دیکود شده است.

لازم به ذکر است که خروجی این تابع، رشته باینری است و با استفاده از تابع `bin2str`، استرینگ ارسال شده را بازسازی می‌کنیم. تابع `bin2str` تعدادی بیت آخر رشته که باعث می‌شود طول رشته بر 5 بخش‌پذیر نباشد را دور می‌ریزد. نحوه تست decoding در تابعی به نام `test` انجام می‌پذیرد و به صورت زیر است:

```
function result = test(str, bitrates, noise, mapset)
    bin_send = str2bin(str, mapset);
    result = cell(length(bitrates), 1);

    for i = 1:length(bitrates)
        bitrate = bitrates(i);
        signal_send = coding_amp(bin_send, bitrate);
        signal_receive = signal_send +
                         noise * randn(size(signal_send));
        bin_receive = decoding_amp(signal_receive, bitrate);
        str_receive = bin2str(bin_receive, mapset);
        result{i} = ['Recieved (bitrate=', num2str(bitrate), ',
noise=', num2str(noise), '): ', str_receive];
    end

end
```

خروجی تابع نیز به صورت زیر است:

```
Recieved (bitrate=1, noise=0): signal
Recieved (bitrate=2, noise=0): signal
Recieved (bitrate=3, noise=0): signal
```

5. اضافه کردن نویز به سیگنال ارسالی

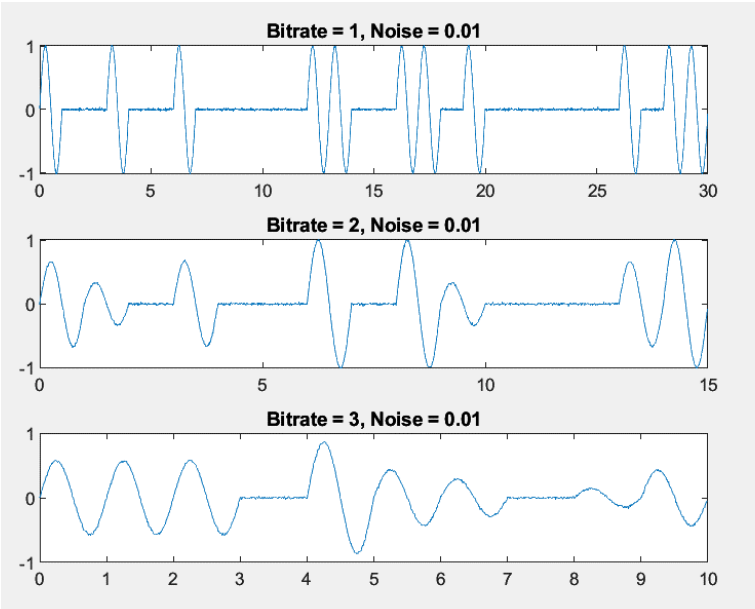
برای اینکه واریانس نویز برابر با 0.0001 شود، باید عدد 0.01 در خروجی تابع `randn` ضرب شود. در نتیجه تابع test را به این صورت فراخوانی می‌کنیم:

```
str = 'signal';
bitrates = 1:3;
noise = 0.01;
result = test(str, bitrates, noise, mapset);
print_result(result)
```

خروجی به صورت زیر خواهد بود:

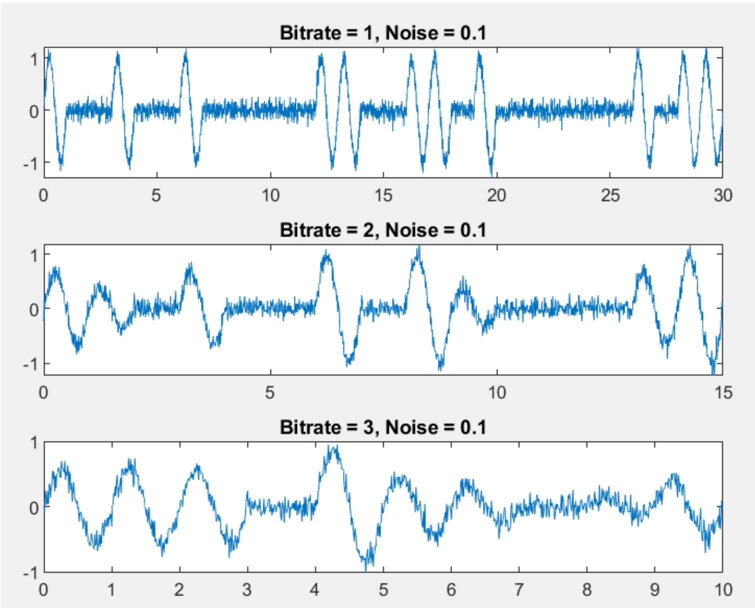
```
Recieved (bitrate=1, noise=0.01): signal
Recieved (bitrate=2, noise=0.01): signal
Recieved (bitrate=3, noise=0.01): signal
>>
```

همانطور که مشاهده می‌شود، با این مقدار نویز همچنان می‌توان رشته را به طور کاملا صحیح بازسازی کرد. برای درک اینکه چه مقدار نویز به سیگنال اضافه شده است، نمودار سیگنال پس از اضافه کردن نویز را رسم می‌کنیم:



6. افزایش نویز

- ابتدا مقدار نویز را از 0.01 به 0.1 افزایش می‌دهیم، نمودار سیگنال‌ها به صورت زیر خواهد بود:

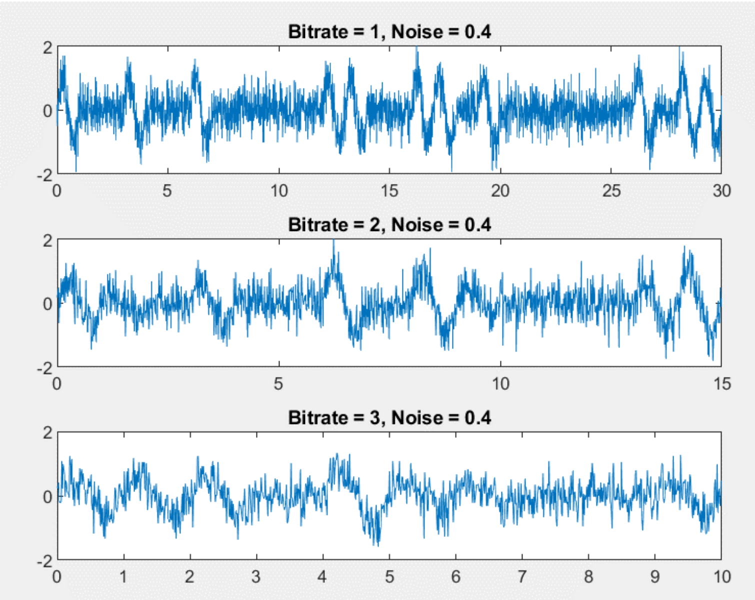


خروجی تابع نیز به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=0.1): signal
Recieved (bitrate=2, noise=0.1): signal
Recieved (bitrate=3, noise=0.1): signal
>>
```

همانطور که مشاهده می‌شود، تابع decoding همچنان می‌تواند رشته ارسال شده را به درستی تشخیص دهد.

- حال مقدار نویز را به 0.4 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

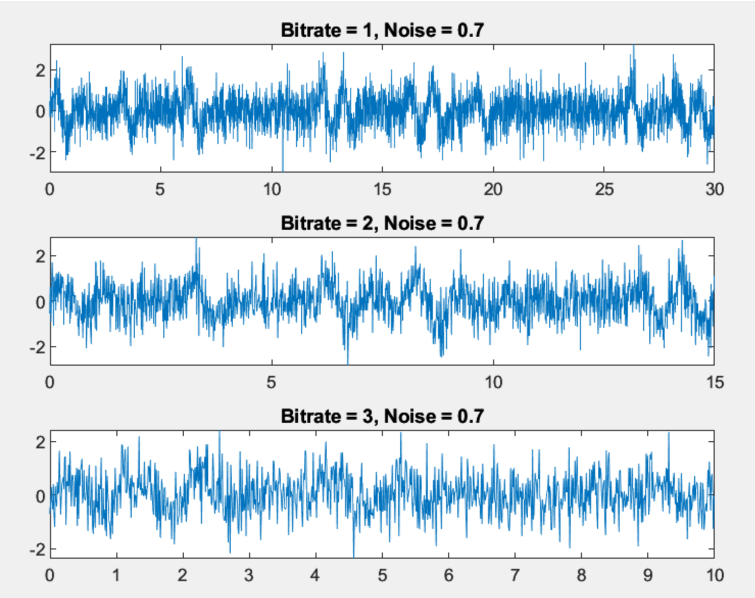


خروجی تابع نیز به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=0.4): signal
Recieved (bitrate=2, noise=0.4): signal
Recieved (bitrate=3, noise=0.4): rygnac
```

تابع با مقدار bitrate = 3 نتوانسته رشته را به درستی بازسازی کند، اما با مقادیر 1 و 2 همچنان به درستی کار می‌کند.

- حال مقدار نویز را به 0.7 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

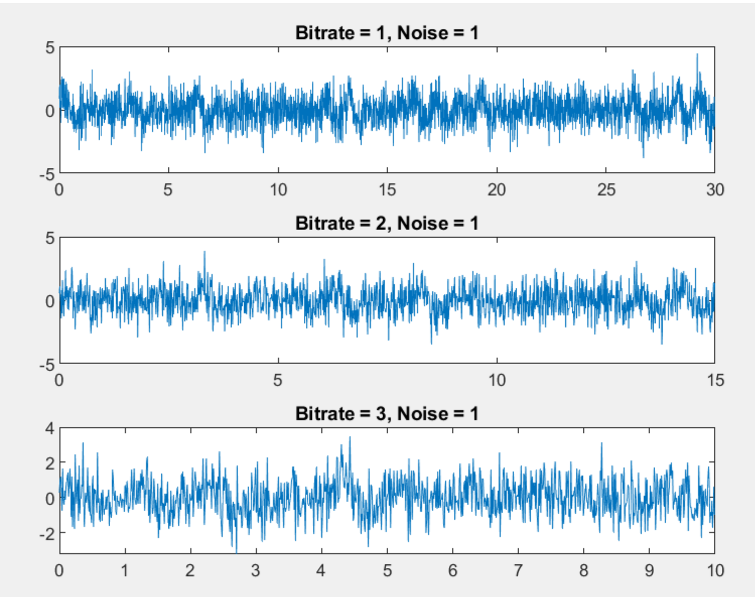


خروجی تابع به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=0.7): signal
Recieved (bitrate=2, noise=0.7): sfgnal
Recieved (bitrate=3, noise=0.7): sknjqc
>>
```

مشاهده می‌شود که در این حالت bitrate = 2 هم نتوانسته به درستی عمل کند اما bitrate = 1 همچنان صحیح عمل می‌کند.

- مقدار نویز را به 1 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

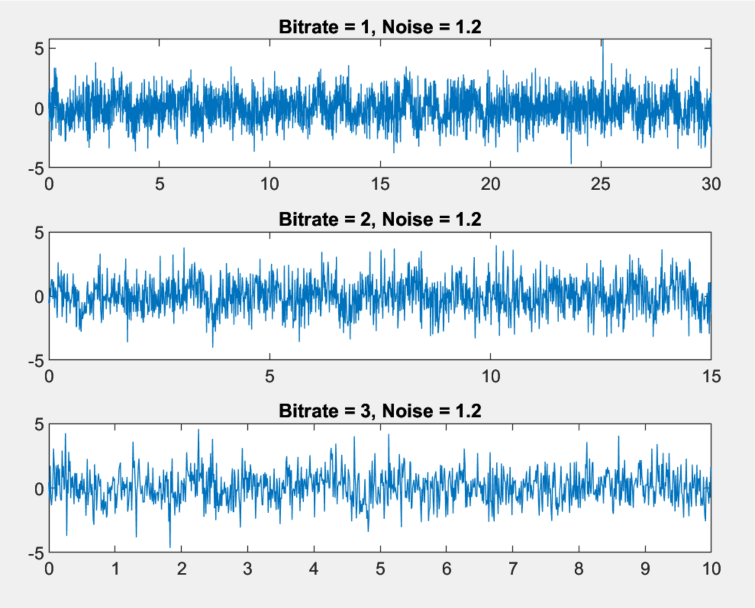


خروجی تابع نیز به صورت زیر است:

```
Recieved (bitrate=1, noise=1): signal
Recieved (bitrate=2, noise=1): signap
Recieved (bitrate=3, noise=1): n oqct
>>
```

مشاهده می‌شود که bitrate = 1 همچنان به درستی عمل می‌کند.

- حال مقدار نویز را به 1.2 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:



خروجی تابع نیز به صورت زیر است:

```
Recieved (bitrate=1, noise=1.2): cignak
Recieved (bitrate=2, noise=1.2): keczak
Recieved (bitrate=3, noise=1.2): nudjcr
```

مشاهده می‌شود که در این حالت حتی bitrate = 1 هم پاسخ‌گو نبوده و نمی‌توان رشته را بازسازی کرد.

در کل، bitrate = 1 از سایر بیت‌ریت‌ها نسبت به نویز مقاوم‌تر بوده و در سطح نویزهایی که سیگنال دریافتی بیت‌ریت‌های بالاتر خراب می‌شدند، بیت‌ریت 1 هنوز جواب درستی می‌گرفته.

همانطور که در مقدمه گفته شد، بدیهی‌ست که با افزایش مقدار bitrate، مقاومت نسبت به افزایش نویز کاهش پیدا می‌کند.

برای اثبات این حرف تابعی به نام `fixed_noise_error` به شکل زیر نوشته شد که به ازای یک مقدار ثابت برای نویز (در این بخش 0.5)، 1000 بار تست را اجرا می‌کنیم و درصد خطای هر بیت‌ریت را محاسبه می‌کنیم.

```
function error = fixed_noise_error(str, bitrate, noise, mapset, 
char_bin_len)
    bin_send = str2bin(str, mapset);
    signal_send = coding_amp(bin_send, bitrate);

    errors = 0;
    test_count = 1000;
    total_parts_count = test_count * ceil(length(str) * 
char_bin_len / bitrate);

    for i = 1:test_count
        signal_receive = signal_send + noise * 
randn(size(signal_send));
        bin_receive = decoding_amp(signal_receive, bitrate);
        for j = 1:bitrate:length(bin_send) - bitrate
            if ~strcmp(bin_send(j:j + bitrate - 1), 
bin_receive(j:j + bitrate - 1))
                errors = errors + 1;
            end
        end

        % Check last part
        if length(bin_send) - j > 0
            if ~strcmp(bin_send(j + bitrate:end), bin_receive(j + 
bitrate:end))
                errors = errors + 1;
            end
        end
    end

    error = errors * 100 / total_parts_count;
end
```

خروجی تابع به ازای هر بیت‌ریت به صورت زیر است:

```
Error (bitrate=1, noise=0.5): 0%
Error (bitrate=2, noise=0.5): 10.58%
Error (bitrate=3, noise=0.5): 36.05%
```

همانطور که مشاهده می‌شود، با افزایش مقدار بیت‌ریت، مقاوت در برابر نویز به مراتب کاهش پیدا می‌کند و درصد خطا افزایش می‌یابد.

7. آستانه مقاوم بودن به نویز

برای به دست آوردن آستانه نویزی که bitrateهای مختلف به آن مقاوم می‌مانند، یک تابع نوشته شد که مقدار نویز را با قدم‌های 0.02 افزایش داده و بررسی می‌کند که آیا با 100 بار ارسال یک پیام (که اینجا همان `signal` در نظر گرفته شده) همه را درست دریافت می‌کند یا خیر.

```
function thold = noise_threshold(str, bitrate, mapset)
    bin_send = str2bin(str, mapset);
    signal_send = coding_amp(bin_send, bitrate);

    thold = 2;
    nStep = 0.02;

    for noise = nStep:nStep:2
        for i = 1:100
            signal_receive = signal_send +
                             noise * randn(size(signal_send));
            bin_receive = decoding_amp(signal_receive, bitrate);
            str_receive = bin2str(bin_receive, mapset);
            if ~strcmp(str, str_receive)
                thold = noise - nStep;
                return
            end
        end
    end
end
```

نتیجه اجرای تابع به ازای 3 بیت‌ریت:

```
Noise threshold (bitrate=1): 0.74
Noise threshold (bitrate=2): 0.26
Noise threshold (bitrate=3): 0.14
```

طبق مقادیر خروجی، مقدار تقریبی بیشترین واریانس نویز برای 3 بیت‌ریت به ترتیب 0.55، 0.07 و 0.02 می‌باشد.

8. راهکار مقاوم‌سازی بیت‌ریت به نویز

برای اینکه افزایش bitrate موجب خراب شدن داده دریافتی گیرنده توسط نویز نشود و به آنها مقاوم بماند، باید دامنه سیگنال نیز افزایش بیابد. در این مثال، برای همه بیت‌ریت‌ها دامنه سیگنال 1 در نظر گرفته شده بود. در صورت افزایش دامنه، فاصله‌های در نظر گرفته شده برای ضرایب سینوس بیشتر شده و نسبت به نویز حساسیت کمتری ایجاد می‌شود.

بخش چهارم

1. ساخت Mapset

در این بخش از همان Mapset ساخته شده در بخش سوم استفاده شده است.

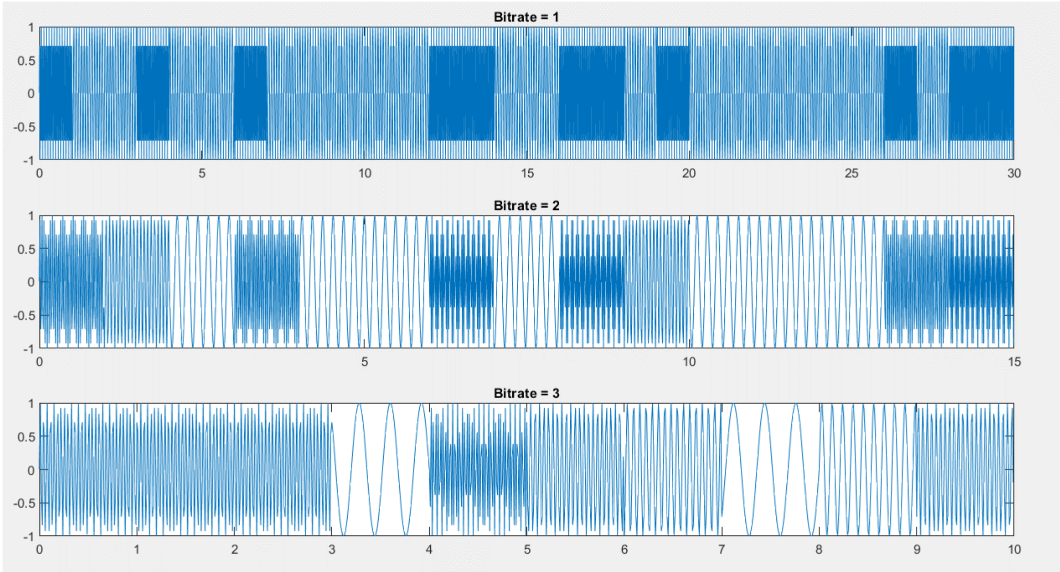
2. تابع coding_freq

این تابع با شناسه زیر در فایل coding_freq.m قرار گرفته است. باقی موارد مشابه تابع `coding_amp` است که پیش‌تر توضیح داده شده است.

```
function signal = coding_freq(bin_msg, bitrate)
```

3. خروجی تابع coding_freq

خروجی این تابع به ازای مقادیر 1، 2 و 3 برای bitrate در تصویر زیر مشخص شده است:



4. تابع decoding_freq

این تابع با شناسه زیر در فایل decoding_freq.m قرار گرفته است:

```
function binary = decoding_freq(signal, bitrate)
```

برای انجام correlation، تبدیل فوریه 100 سمپل سینوس را به دست آورده و در نقطه‌ای که پیک می‌کند، فرکانس را داریم. سپس نزدیک‌ترین نقطه در لیست فرکانس‌های بیت‌ریت مورد نظر را به فرکانس فوریه حساب کرده که اندیس آن عدد دیکود شده است.

خروجی این تابع همانند `decoding_amp` رشته باینری است و با استفاده از `bin2str` استرینگ بازسازی می‌شود. با استفاده از تابع `test` مشابه قسمت سوم، می‌توان خروجی را بررسی کرد:

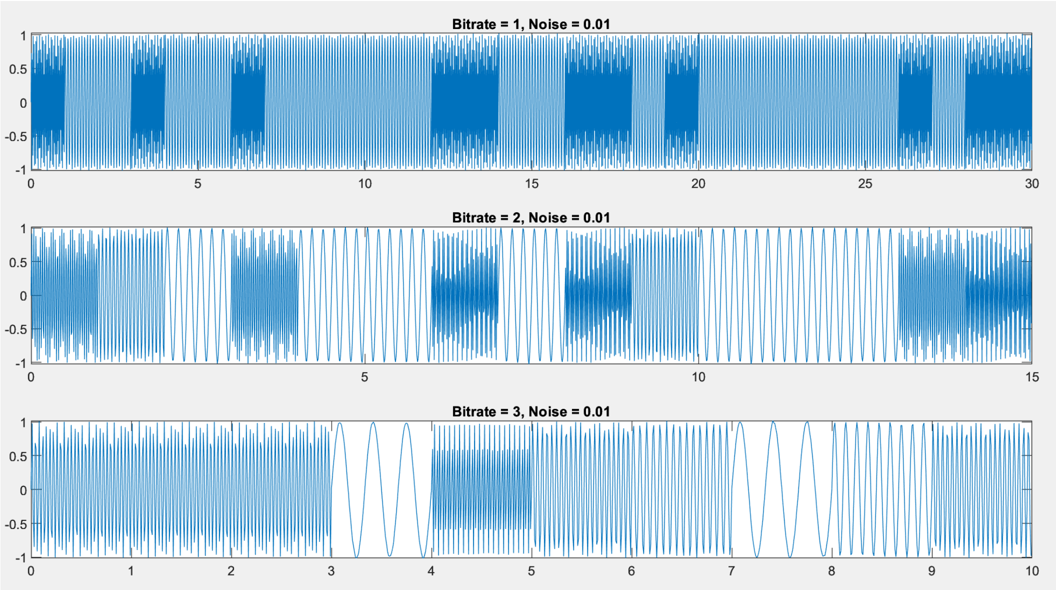
```
Recieved (bitrate=1, noise=0): signal
Recieved (bitrate=2, noise=0): signal
Recieved (bitrate=3, noise=0): signal
```

5. اضافه کردن نویز به سیگنال ارسالی

برای اعمال نویز با واریانس 0.0001 باید عدد 0.01 در خروجی `randn` ضرب شود. تابع `test` فراخوانی شده و نتیجه آن به شکل زیر است:

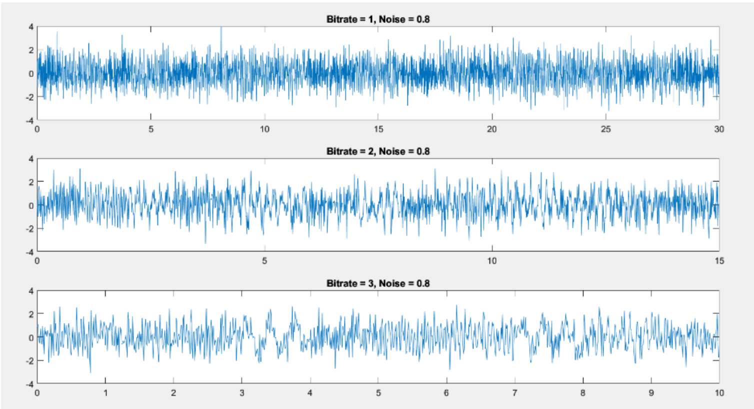
```
Recieved (bitrate=1, noise=0.01): signal
Recieved (bitrate=2, noise=0.01): signal
Recieved (bitrate=3, noise=0.01): signal
>>
```

با این مقدار نویز، رشته همچنان به صورت کامل بازسازی می‌شود. نمودار سیگنال پس از اضافه کردن نویز:



6. افزایش نویز

- ابتدا مقدار نویز را از 0.01 به 0.8 افزایش می‌دهیم، نمودار سیگنال‌ها به صورت زیر خواهد بود:

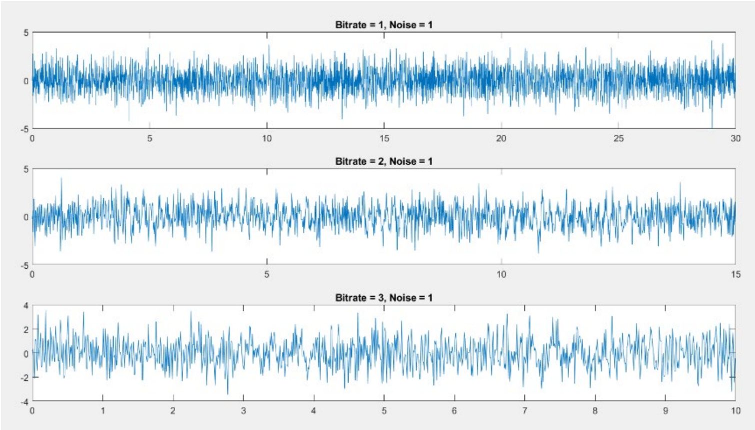


خروجی تابع نیز به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=0.8): signal
Recieved (bitrate=2, noise=0.8): signal
Recieved (bitrate=3, noise=0.8): signal
```

همانطور که مشاهده می‌شود، تابع decoding همچنان می‌تواند رشته ارسال شده را به درستی تشخیص دهد.

- حال مقدار نویز را به 1 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

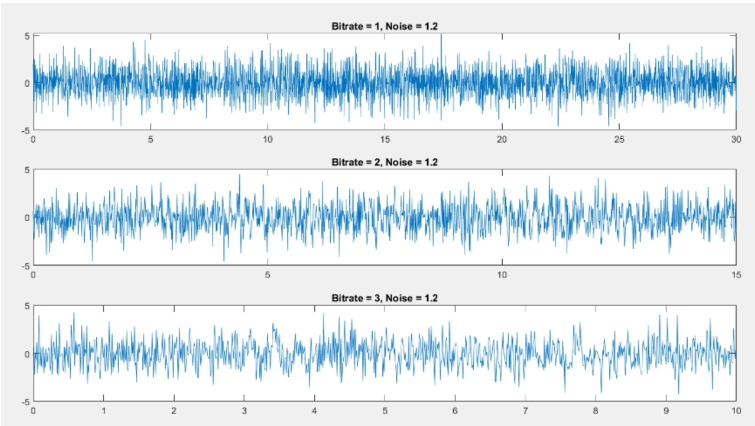


خروجی تابع نیز به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=1): signal
Recieved (bitrate=2, noise=1): signal
Recieved (bitrate=3, noise=1): signal
```

تابع همچنان به درستی عمل می‌کند.

- حال مقدار نویز را به 1.2 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

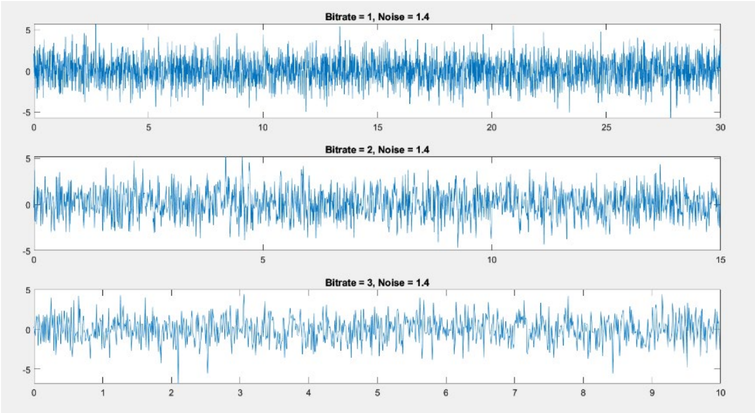


خروجی تابع به صورت زیر خواهد بود:

```
Recieved (bitrate=1, noise=1.2): signal
Recieved (bitrate=2, noise=1.2):  ignal
Recieved (bitrate=3, noise=1.2): signal
```

مشاهده می‌شود که در این حالت bitrate = 2 نتوانسته به درستی عمل کند اما مقادیر 1 و 3 درست عمل کرده‌اند. در واقع میزان مقاومت 3 بیت‌ریت در برابر نویز تقریبا یکسان است و دلیل عملکرد درست و غلط آن‌ها در یک نویز خاص، رندوم بودن نویز است.

- مقدار نویز را به 1.4 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:

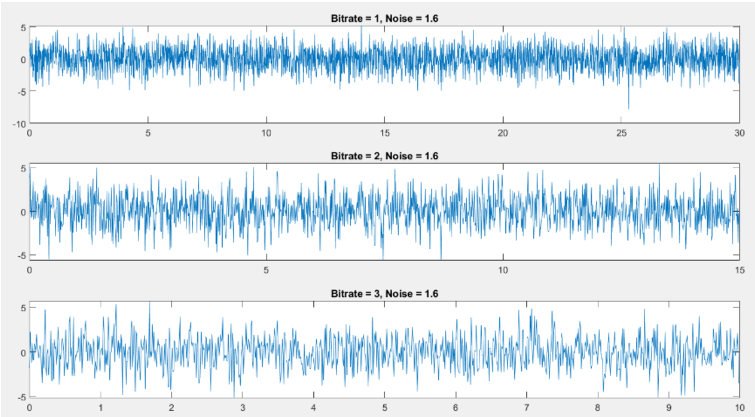


خروجی تابع نیز به صورت زیر است:

```
Recieved (bitrate=1, noise=1.4): siwnal
Recieved (bitrate=2, noise=1.4): signap
Recieved (bitrate=3, noise=1.4): signal
```

مشاهده می‌شود که bitrate = 1 هم نتوانسته به درستی عمل کند اما بیت‌ریت 3 درست عمل کرده است.

- حال مقدار نویز را به 1.6 افزایش می‌دهیم. نمودار سیگنال‌ها به صورت زیر خواهد بود:



خروجی تابع نیز به صورت زیر است:

```
Recieved (bitrate=1, noise=1.6): sngnfd
Recieved (bitrate=2, noise=1.6): slgbal
Recieved (bitrate=3, noise=1.6): okgnal
```

مشاهده می‌شود که در این حالت هیچ کدام نتوانسته‌اند پاسخ صحیحی بگیرند.

در این بخش هر 3 مقدار برای بیت‌ریت تا حد خوبی مشابه هم عمل کردند. به همین دلیل برای مشاهده مقدار مقاومت هر کدام در برابر نویز، از تابع `fixed_noise_error` مشابه بخش 3 استفاده می‌کنیم.

خروجی تابع به ازای هر بیت‌ریت برای نویز 1.5 به صورت زیر است:

```
Error (bitrate=1, noise=1.5): 4.6867%
Error (bitrate=2, noise=1.5): 7.12%
Error (bitrate=3, noise=1.5): 8.72%
```

همانطور که مشاهده می‌شود، با افزایش مقدار بیت‌ریت، مقاوت در برابر نویز به مراتب کاهش پیدا می‌کند و درصد خطا افزایش می‌یابد.

در مقایسه با بخش 3 می‌توان دید که مقاومت در برابر نویز در این روش بسیار افزایش پیدا کرده است.

7. آستانه مقاوم بودن به نویز

برای به دست آوردن آستانه نویزی که bitrateهای مختلف به آن مقاوم می‌مانند، از تابع `noise_threshold` مشابه بخش سوم استفاده شد که مقدار نویز را با قدم‌های 0.02 افزایش داده و بررسی می‌کند که آیا با 100 بار ارسال یک پیام (که اینجا همان `signal` در نظر گرفته شده) همه را درست دریافت می‌کند یا خیر.

نتیجه اجرای تابع به ازای 3 بیت‌ریت:

```
Noise threshold (bitrate=1): 0.98
Noise threshold (bitrate=2): 0.98
Noise threshold (bitrate=3): 0.94
```

طبق مقادیر خروجی، مقدار تقریبی بیشترین واریانس نویز برای 3 بیت‌ریت به ترتیب 0.96، 0.96 و 0.88 می‌باشد.

8. راهکار مقاوم‌سازی بیت‌ریت به نویز

افزایش بیت‌ریت باعث می‌شود فرکانس‌های انتخابی بسیار نزدیک به هم شوند و در این صورت با افزایش مقدار نویز، ممکن است فرکانس را اشتباه تشخیص دهیم. برای جبران این مورد می‌توانیم فرکانس نمونه‌برداری را افزایش دهیم که در این صورت محدوده فرکانس‌های قابل تخصیص به هر مقدار باینری افزایش می‌یابد و فرکانس‌ها از هم دورتر می‌شوند که این مورد باعث افزایش دقت برنامه می‌شود. در واقع با این کار پهنای باند را افزایش داده‌ایم.