شرح مختصر پروژه

در این پروژه هدف ما این است که از سیگنال AM دریافتی از آنتن رادیو صدای کانال‌های رادیویی مختلف را استخراج کنیم.

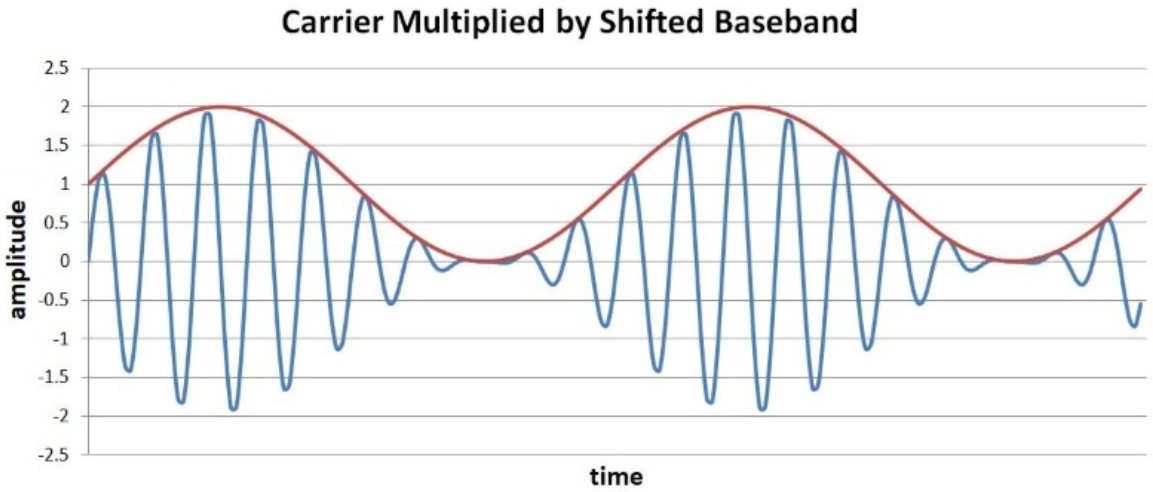
برای این کار ما یک فایل متنی در اختیار داریم که حاوی ۱۰ ثانیه سیگنال دریافتی از آنتن است.

نحوه‌ی تلفیق سیگنال AM

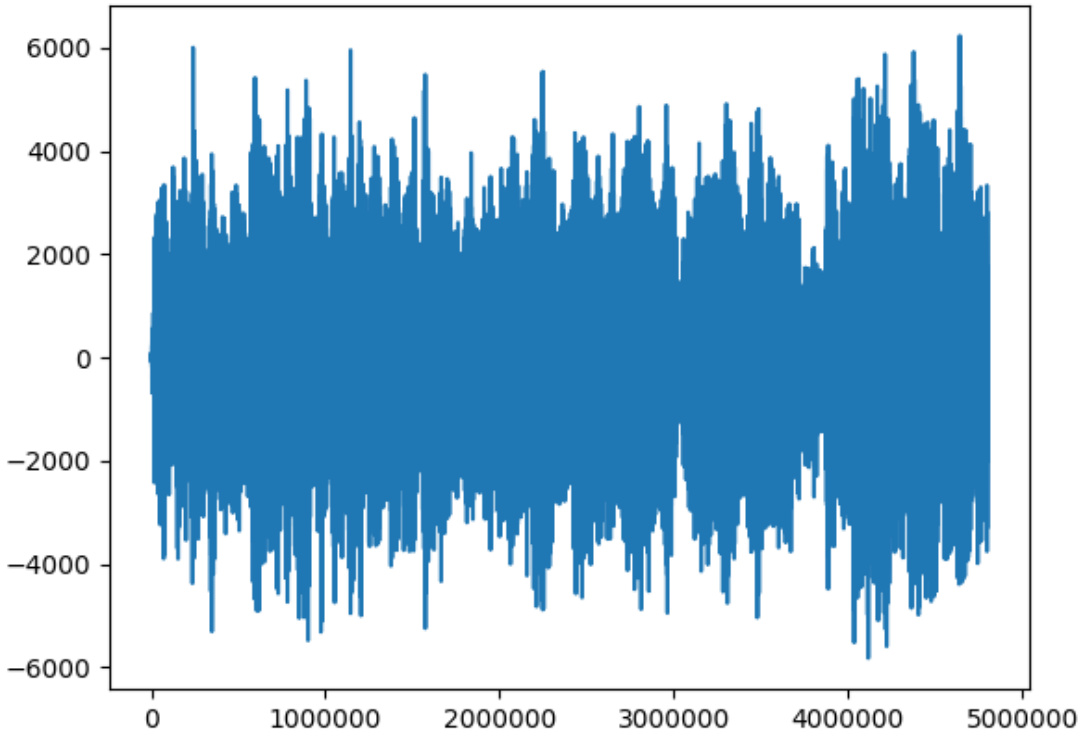
سیگنال AM از ضرب دو سیگنال ساخته می‌شود: سیگنال حامل (Carrier) و سیگنال داده (baseband). در سیگنال حاصله فرکانس سیگنال حامل تغییری نمی‌کند، امّا با توجّه به سیگنال داده، مقدار دامنه (amplitude) به صورت مداوم تغییر می‌کند.

عموماً سیگنال حامل یک سیگنال متناوب سینوسی در نظر گرفته می‌شود.

شکل زیر یک مثال از این فرآیند است که از این آدرس برداشته شده است (سیگنال یک واحد به سمت بالا شیفت پیدا کرده است. رنگ قرمز: سیگنال حاوی داده - رنگ آبی: سیگنال AM).



سیگنالی که برای این پروژه در اختیار داریم در بازه‌ی زمان شکل زیر را دارد:



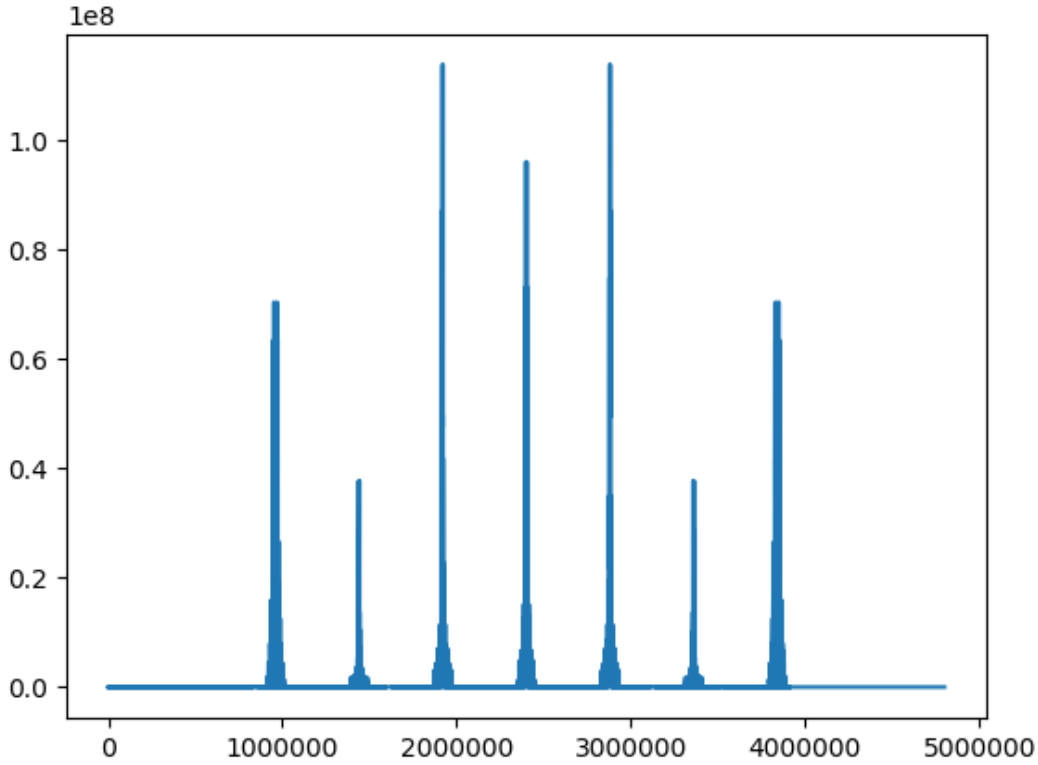
نحوه‌ی استخراج کانال مورد نظر

ما می‌خواهیم که سیگنال حاوی صوت کانال مدّنظرمان را از درون سیگنال AM ورودی استخراج کنیم. برای این کار ما باید ۳ مرحله را پشت سر بگذاریم:

۱- جداکردن کانال مورد نظر از بقیه‌ی داده‌ها

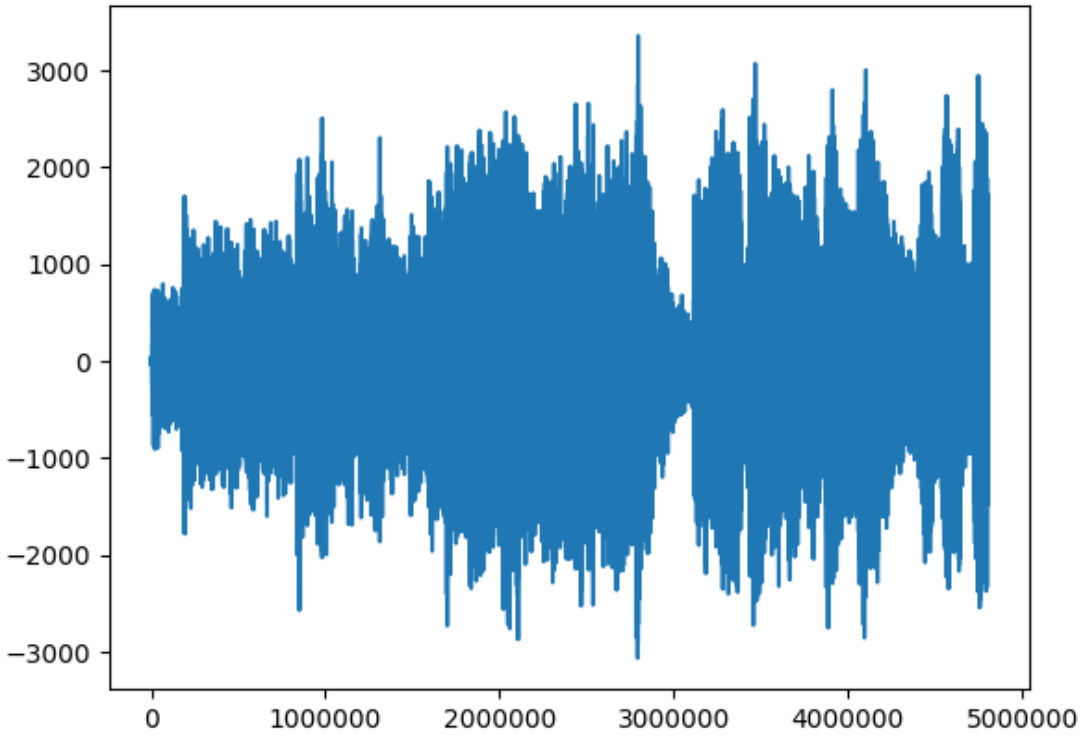
اوّلین کاری که می‌خواهیم انجام بدهیم این است که با یک فیلتر میان‌گذر، کانال مورد نظرمان را از دیگر کانال‌ها جداکنیم.

اگر تبدیل فوریه‌ی سیگنال دریافتی را مشاهده کنیم، می‌بینیم که چنین شکلی دارد:

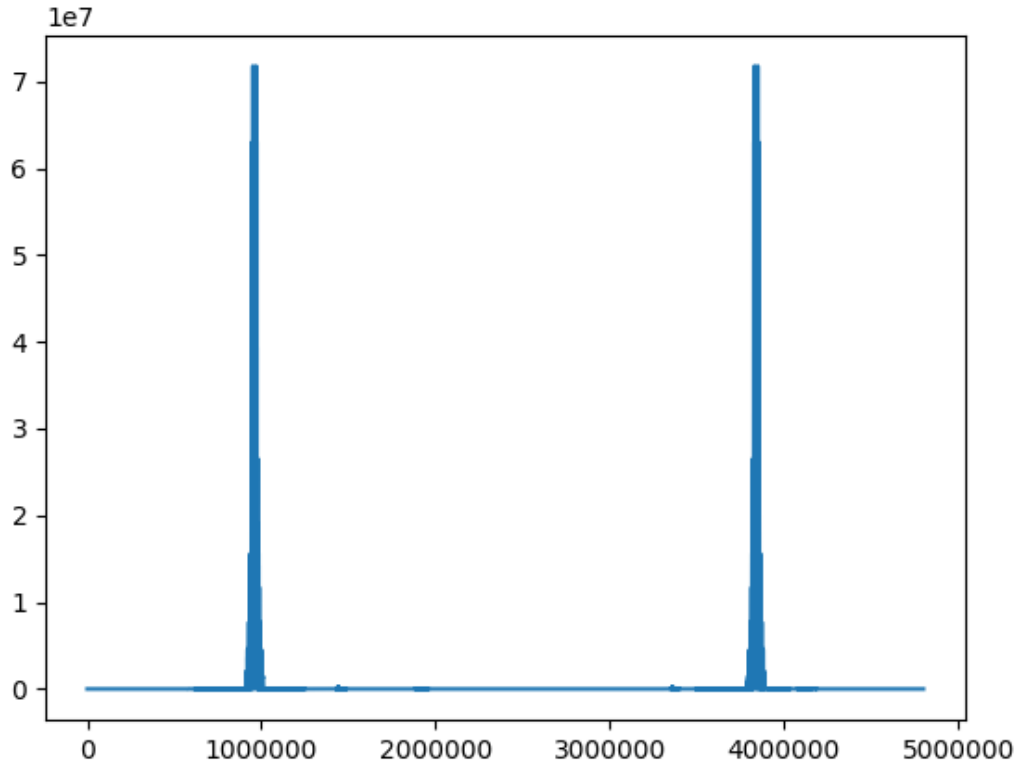


همانطوری که مشاهده می‌شود ما در اینجا سیگنال‌های مربوط به کانال‌های مختلف را داریم. حالا با فیلتر میان‌گذر تمامی داده‌ها، به جز بازه‌ی فرکانسی‌ای که صوت آن را می‌خواهیم، حذف می‌کنیم.

برای مثال اگر کانالی با فرکانس ۹۶KHz را بخواهیم، خروجی فیلتر میان‌گذر مانند شکل زیر خواهد بود:



حالا تبدیل فوریه‌ی این سیگنال را می‌بینیم:



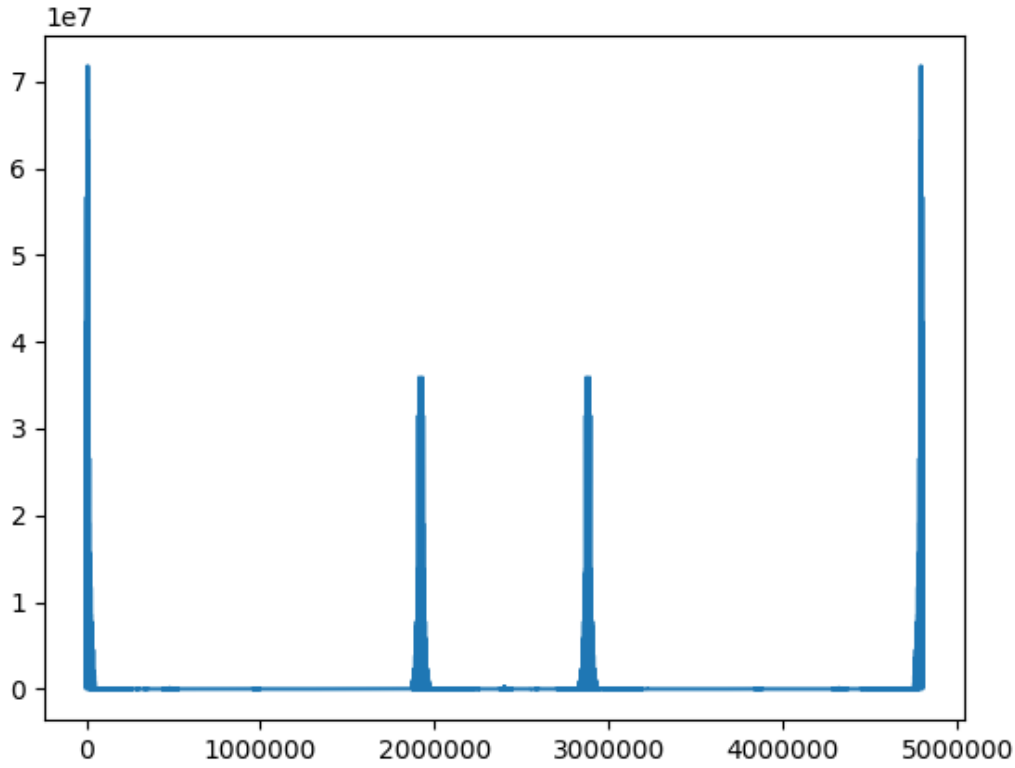
همانطوری که مشاهده می‌شود تنها دو بخش مربوط به خروجی fft این کانال باقی‌مانده‌اند و دیگر کانال‌ها حذف شده اند.

انتقال سیگنال

پس از جداسازی بخش مربوط به کانال دلخواه، باید آن سیگنال را در بازه‌ی فرکانسی انتقال بدهیم تا وارد محدوده‌ی مورد نظرمان بشود.

از آنجایی که ما در تمام بخش‌های این پروژه در بازه‌ی زمان کار می‌کنیم، برای این انتقال سیگنال خروجی مرحله‌ی قبل را در سیگنال حامل ضرب می‌کنیم. چون ضرب در حوزه‌ی زمان برابر با انتقال فرکانسی است.

خروجی انتقال سیگنال مرحله‌ی قبل در حوزه‌ی فرکانس شکل زیر خواهد بود:

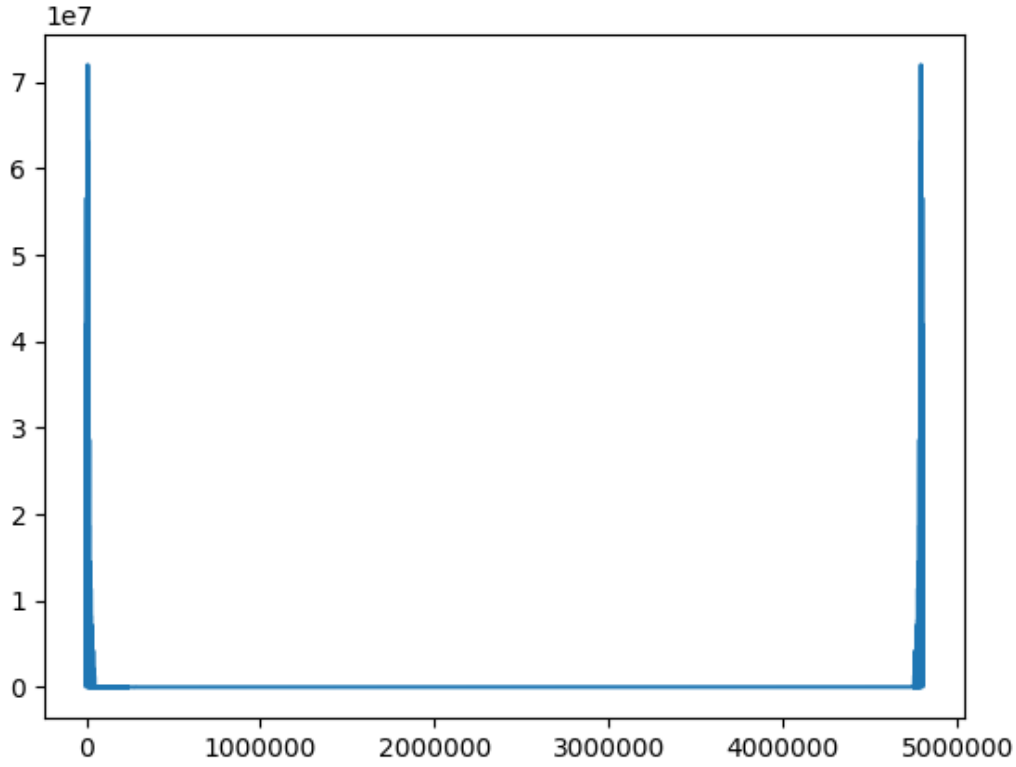


همانطوری که مشاهده می‌شود این کار علاوه بر انتقال، باعث می‌شود که سیگنال مورد نظر دوبار تکرار شود.

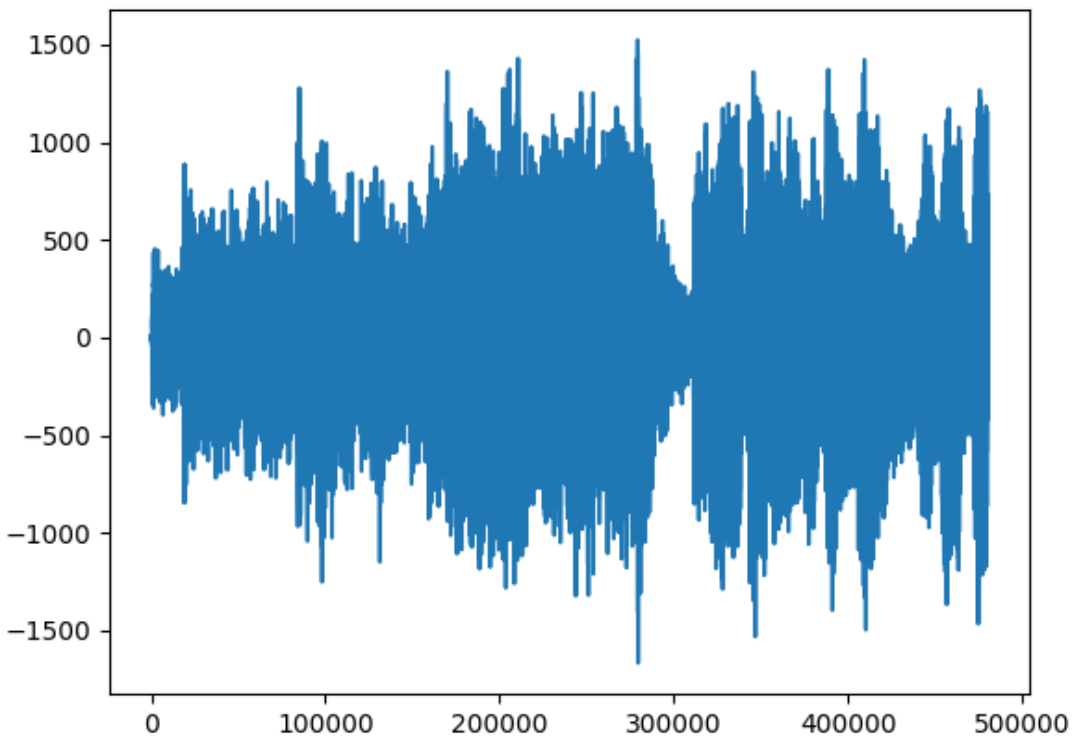
۳- فیلترکردن بخش مورد نظر

در مرحله‌ی سوم ما با یک فیلتر پایین‌گذر تنها تکرار ابتدایی را که داده‌های مورد نظرمان در بازه‌ی مناسب قرار دارند نگه‌می‌دارد و قسمت بعدی را حذف می‌کند.

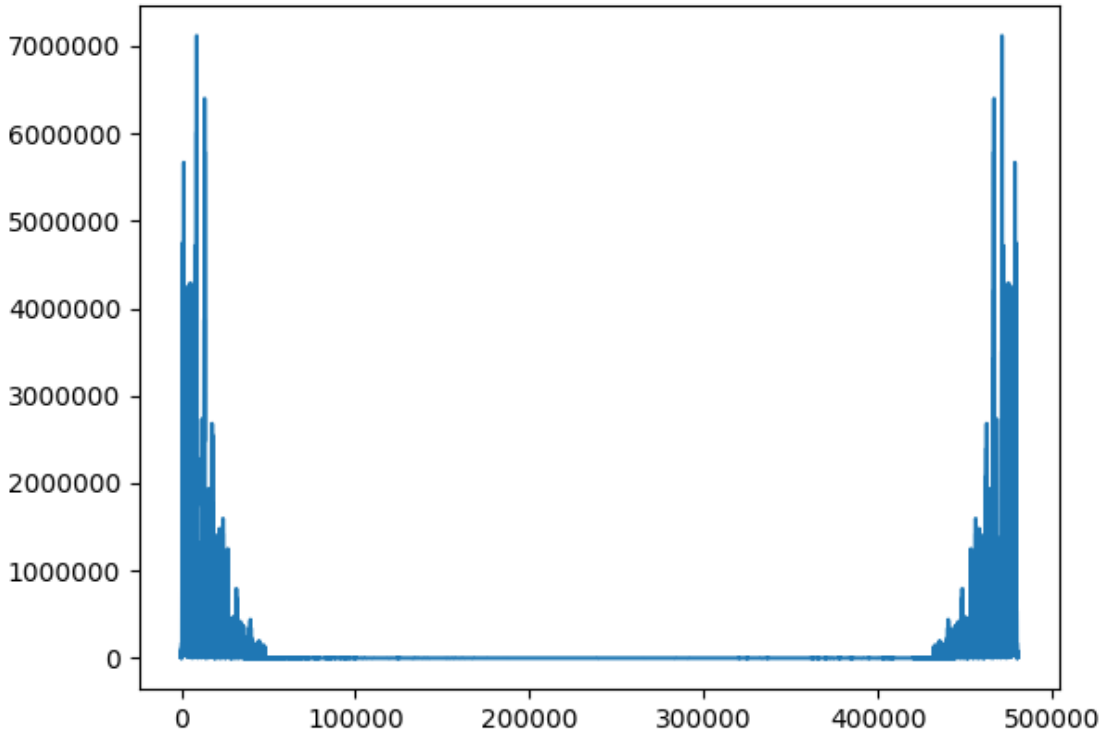
پس از اعمال این فیلتر، سیگنال باقی‌مانده شکل زیر را خواهد داشت:



حالا می‌توانیم خروجی را downsample کنیم. در این پروژه ما با ضریب ده این کار را انجام داده‌ایم. خروجی نهایی در حوزه‌ی زمان شکل زیر را خواهد داشت:



و اگر تبدیل فوریه‌ی آن را محاسبه کنیم، حاصل شکل زیر را خواهد داشت:



در پایان سیگنال صوت جداشده را با فرکانس ۴۴۱۰۰Khz درون یک فایل با فرمت *wav* می‌نویسیم.

پیاده‌سازی

در این بخش از گزارش، به توضیح پیاده‌سازی صورت‌گرفته در این پروژه می‌پردازیم.

این پروژه از دو مؤلفه‌ی(component) اصلی تشکیل شده است: **Filters** و **Radio**.

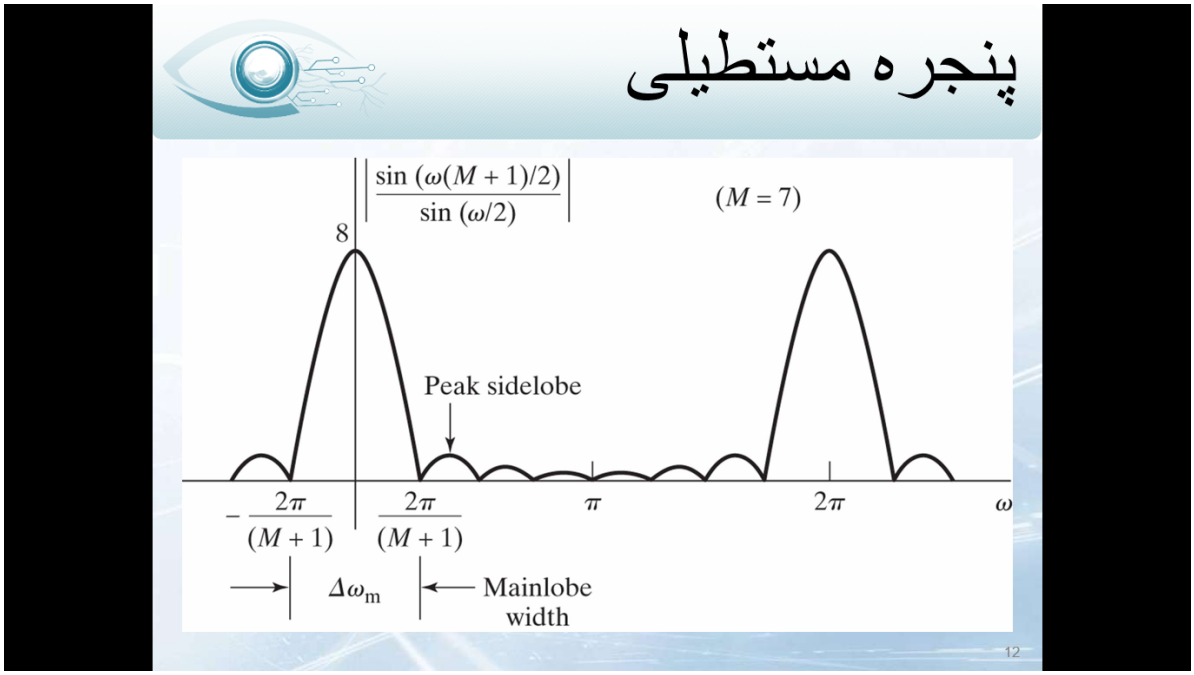
در اینجا شرح مختصر از عملکرد کلّی بخش‌های پیاده‌سازی شده آورده شده است. برای اطّلاعات بیشتر می‌توانید به اسناد و comment های موجود در کد مراجعه کنید.

فیلترها

ما به فیلتر به عنوان یک موجودیت منطقی واحد نگاه می‌کنیم. لذا انواع مختلف فیلترها باید یک فیلتر محسوب شوند (رابطه‌ی IS_A). این موجودیت واحد را کلاسی مجرّد (abstract) با عنوان *BaseFilter* نمایندگی می‌کند.

هر کلاسی که از این این کلاس ارث‌بری می‌کند، باید ویژگی (property) های زیر را پیاده‌سازی کند:

- m
- time
- window
- cutoff_frequency
- start_frequency
- end_frequency
- ideal_filter
- h



ما فیلترها را در حوزه‌ی زمان اعمال می‌کنیم، به همین خاطر برای فیلتر کردن ورودی، باید تابع فیلتر را با سیگنال صوتی دریافت شده convolve کنیم.

تابع فیلتر مطابق شکل زیر است:

$$h[n] = h_d[n]w[n]$$

مقدار w همان پنجره‌ی مؤثّر هر فیلتر است. و مقداری که در آن ضرب می‌شود فیلتر پایین‌گذر ایده‌آل است.

ما ۵ فیلتر مختلف را بر اساس این کلاس پایه پیاده‌سازی کرده‌ایم:

1. Bartlett Filter
2. Blackman Filter
3. Hamming Filter
4. Hann Filter
5. Rectangular

در هر زیرکلاس مقدار `window` و `m` باید به صورت جداگانه تعریف شود، امّا بقیه‌ی ویژگی‌ها و متدها، چون بین همه یکسان هستند، مستقیماً از `BaseFilter` ارث‌بری می‌شوند.

فیلتر Bartlett

در این فیلتر ما مقدار `M` را به صورت زیر محاسبه می‌کنیم:

```
int(np.ceil((8 * np.pi) / (2 * self.delta_w))) + 1
```

در تمامی فیلترها ما مقدار `M` را به صورت یک عدد فرد در خواهیم آورد.

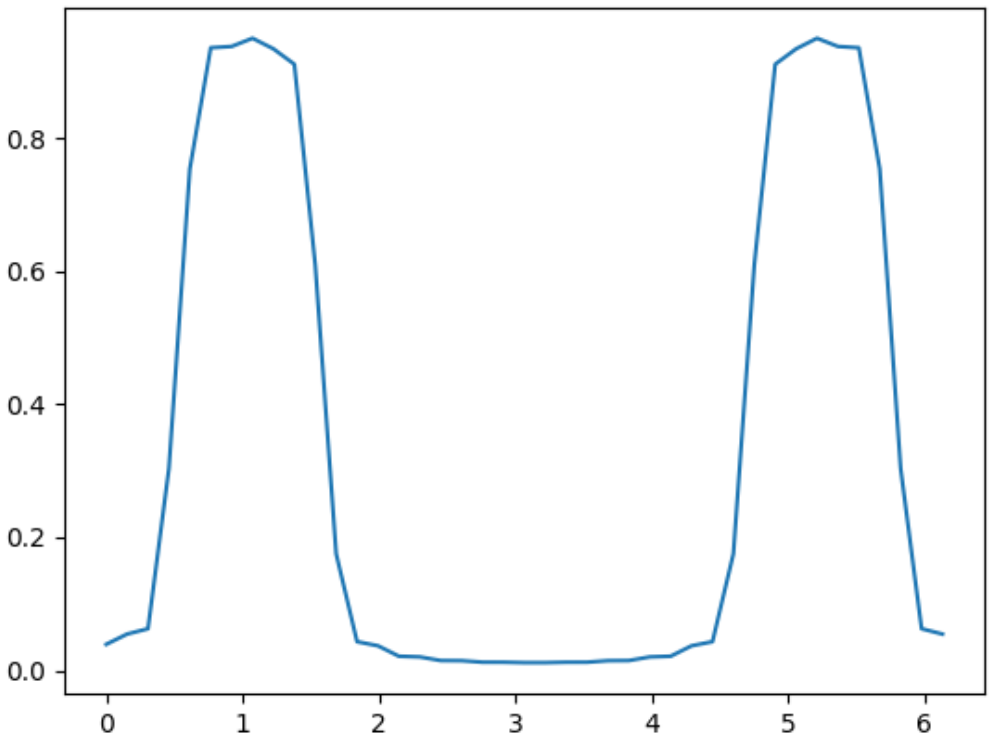
برای ساخت تابع `window` از تابع `bartlett` در ماژول *scipy* استفاده شده است.

برای درک بهتر هر فیلتر، ما مقدار تابع `h` آن‌ها را در حوزه‌ی فرکانس رسم کرده‌ایم.

مقادیر ورودی فیلتر هنگام ساخت تصویر خروجی (هم برای این فیلتر و هم برای فیلترهای دیگر):

پارامتر ورودی	مقدار
delta_w	`np.pi * 0.1`
start_freq	`np.pi/6`
end_freq	np.pi/2

اگر تابع `h` را با استفاده از ماژول *matplotlib* رسم کنیم، تصویر زیر را دریافت خواهیم کرد:



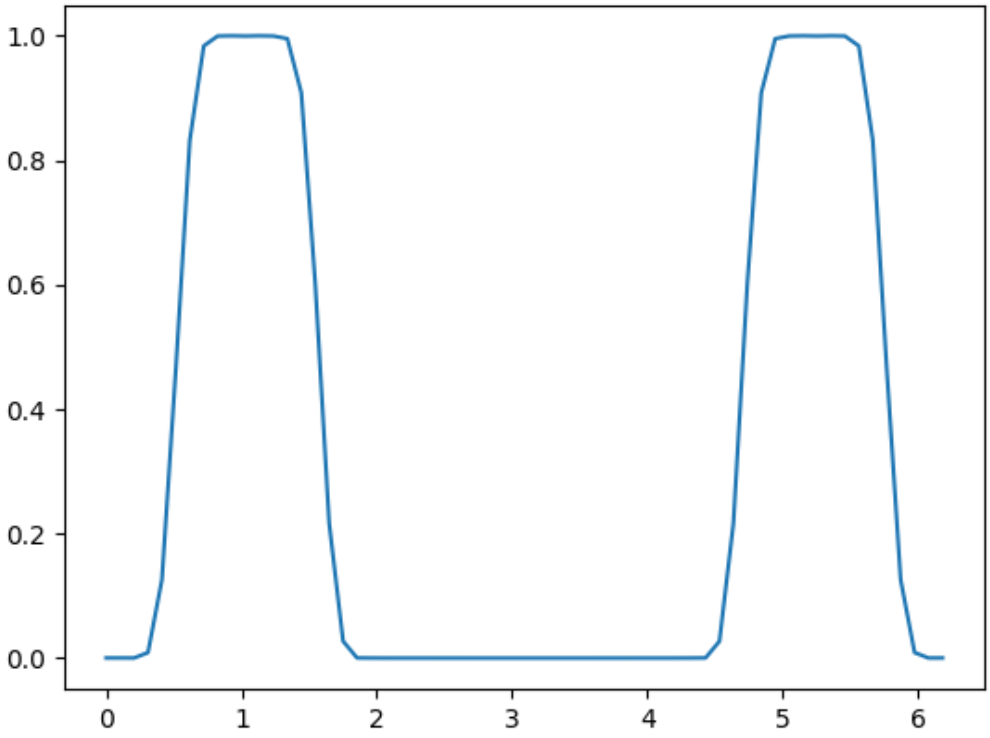
فیلتر Blackman

در این فیلتر ما مقدار `M` را به صورت زیر محاسبه می‌کنیم:

```
int(np.ceil((12 * np.pi) / (2 * self.delta_w))) + 1
```

برای ساخت تابع `window` از تابع `blackman` در ماژول *scipy* استفاده شده است.

اگر تابع `h` را با استفاده از ماژول *matplotlib* رسم کنیم، تصویر زیر را دریافت خواهیم کرد:



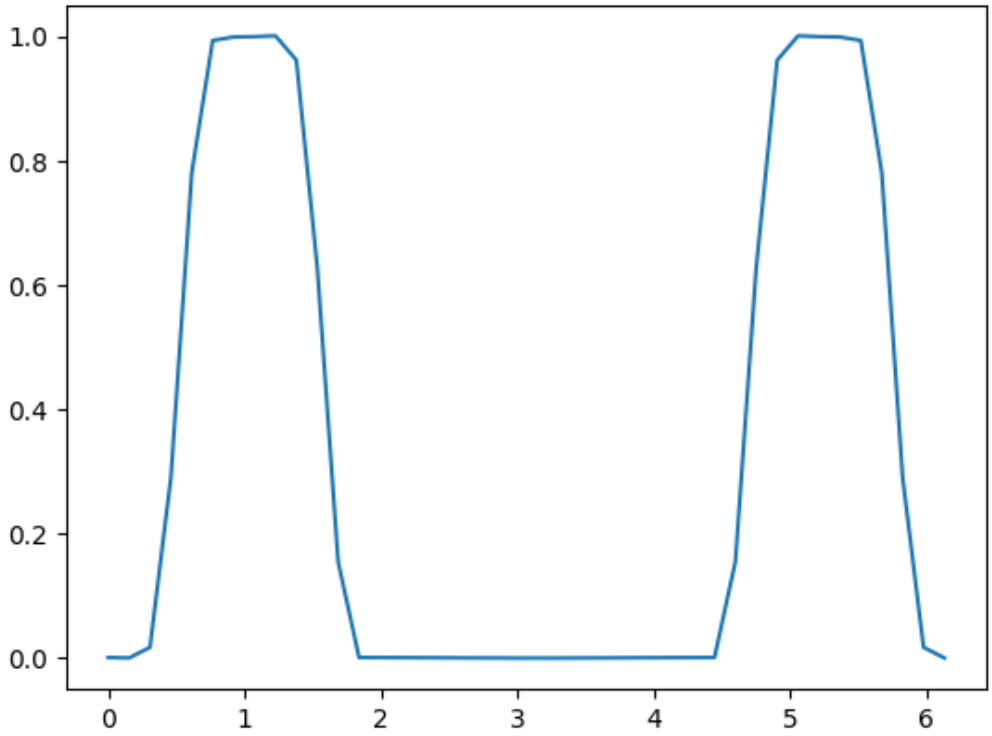
فیلتر Hamming

در این فیلتر ما مقدار `M` را به صورت زیر محاسبه می‌کنیم:

```
int(np.ceil((8 * np.pi) / (2 * self.delta_w))) + 1
```

برای ساخت تابع `window` از تابع `hamming` در ماژول *scipy* استفاده شده است.

اگر تابع `h` را با استفاده از ماژول *matplotlib* رسم کنیم، تصویر زیر را دریافت خواهیم کرد:



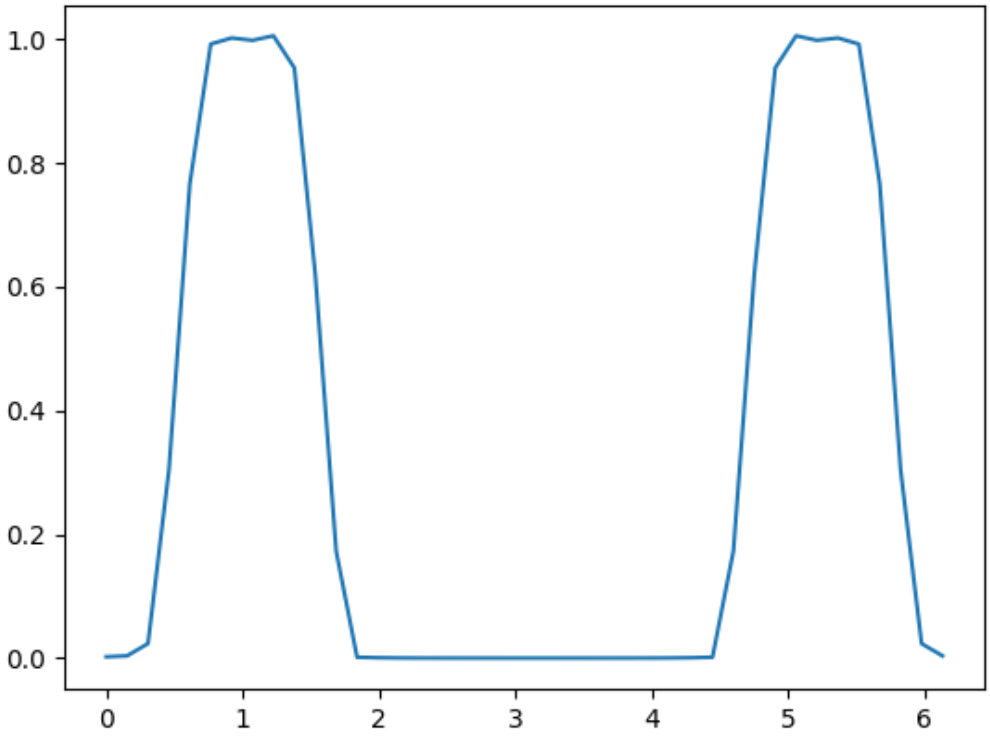
فیلتر Hann

در این فیلتر ما مقدار `M` را به صورت زیر محاسبه می‌کنیم:

```
int(np.ceil((8 * np.pi) / (2 * self.delta_w))) + 1
```

برای ساخت تابع `window` از تابع `hann` در ماژول *scipy* استفاده شده است.

اگر تابع `h` را با استفاده از ماژول *matplotlib* رسم کنیم، تصویر زیر را دریافت خواهیم کرد:



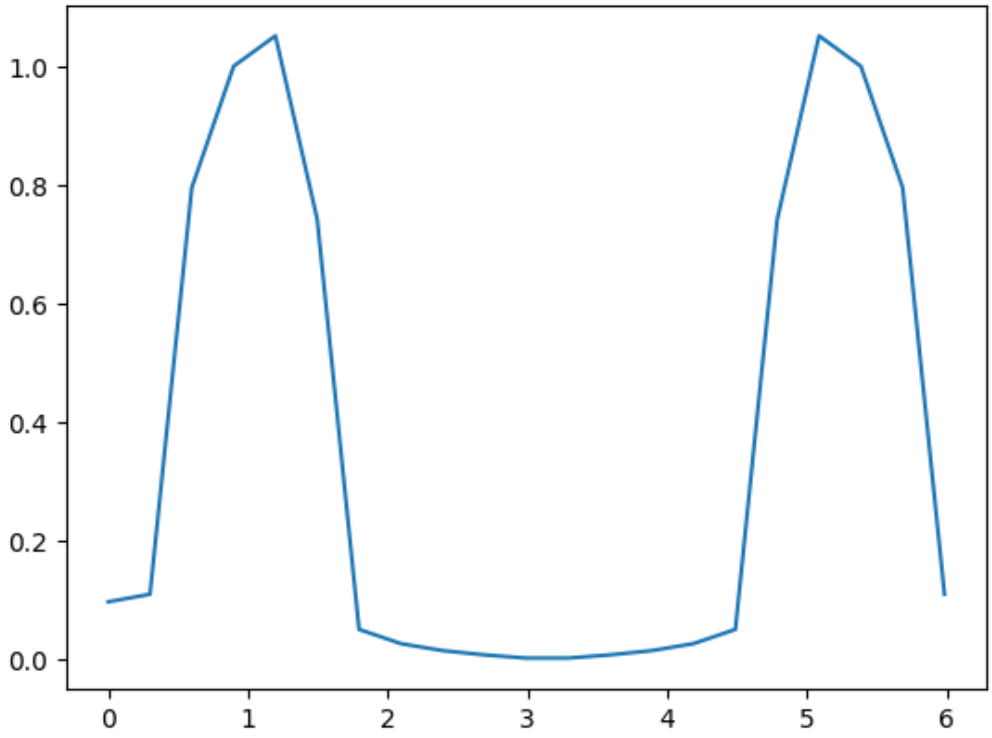
فیلتر Rectangular

در این فیلتر ما مقدار `M` را به صورت زیر محاسبه می‌کنیم:

```
1  int(np.ceil((4 * np.pi) / (2 * self.delta_w)))
```

برای ساخت تابع `window` از تابع `boxcar` در ماژول *scipy* استفاده شده است.

اگر تابع `h` را با استفاده از ماژول *matplotlib* رسم کنیم، تصویر زیر را دریافت خواهیم کرد:



انتخاب بهترین فیلتر باتوجّه به خطای قابل تحمّل

ما با استفاده از تابع `best_filter` بهترین فیلتری را که می‌تواند مقدار خطای قابل پذیرش را تحمّل کند انتخاب می‌کنیم.

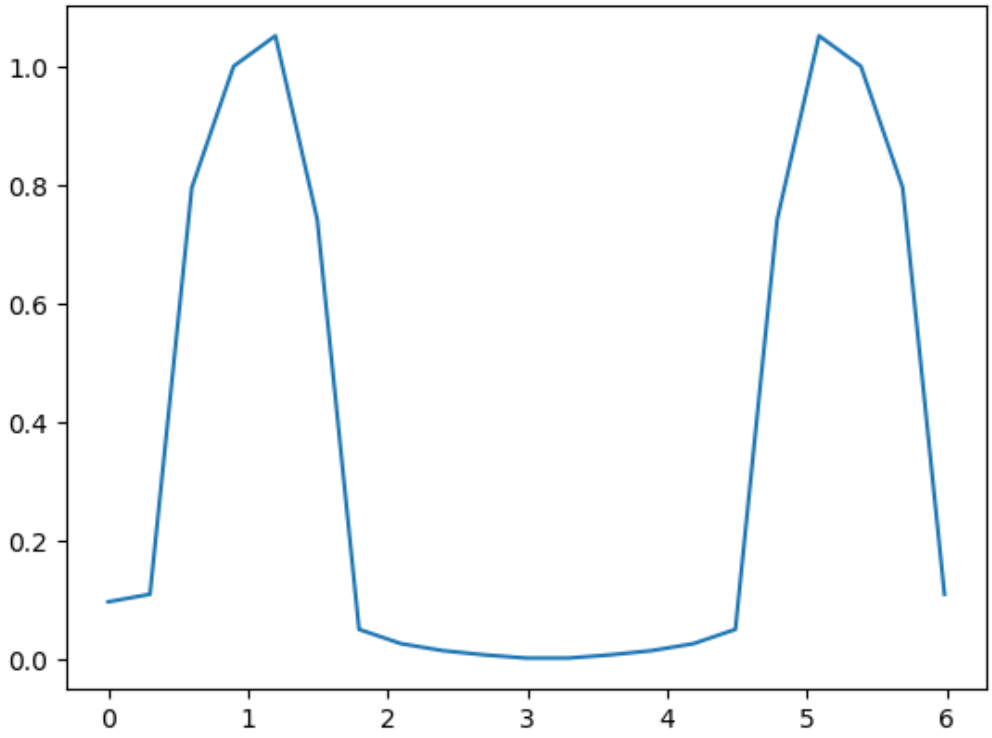
برای این کار لگاریتم خطای قابل تحمّل را محاسبه می‌کنیم و حاصل آن را در ۲۰ ضرب می‌کنیم. بهترین فیلتر، اوّلین فیلتر در جدول زیر است که مقداری کمتر از این عدد داشته باشد.

فیلتر انتخابی	۲۰ × لگارتیم خطا >
Rectangula	-۲۱
Bartlett	-۲۵
Hann	-۴۴
Hamming	-۵۳
Blackman	-۷۴

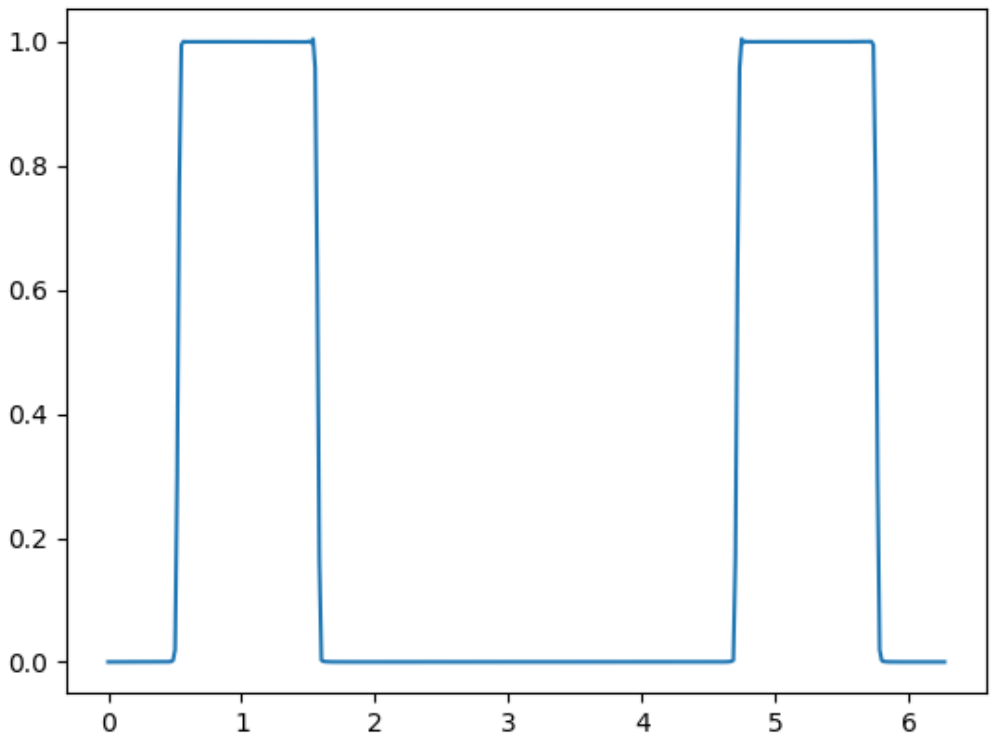
به جای استفاده از این فیلترها می‌توان از فیلتر ایده‌آل هم استفاده کرد، امّا چون تعداد جملات مؤثّر زیاد می‌شود هم مصرف منابع بالا خواهد رفت و هم تأخیر زیاد می‌شود.

ما می‌توانیم با مقدار خطای قابل تحمّل ۰/۱، و استفاده از فیلتر مستطیلی ۲۱ جمله‌ای به خروجی دلخواهمان برسیم.

تصویر زیر فیلتر مستطیلی ۲۱ جمله‌ای است:



در صورتی که اگر از فیلتر Hann با ۴۰۱ جمله هم استفاده کنیم، خروجی تفاوت محسوسی نخواهد داشت.



به همین دلیل توصیه می‌شود که برای مصرف منابع کمتر و گرفتن سریع‌تر خروجی، بازه‌ی خطا را به جای ۰٫۰۱ در Hann روی ۰٫۱ قرار بدهید.

رادیو

مؤلّفه‌ی **Radio** از دو کلاس اصلی تشکیل شده است: *Signal* و *Radio*

سیگنال

هر نمونه از کلاس `Signal` اطّلاعات اصلی یک سیگنال، یعنی مقادیر آن را در حوزه‌ی زمان و نرخ نمونه‌برداری آن را نگهداری می‌کند.

به علاوه با فراخوانی متدهای: `plot_signal_time_domain` و `plot_signal_frequency_domain` می‌توان سیگنال را در حوزه‌های زمان و فرکانس رسم کرد.

برای دریافت تبدیل فوریه‌ی سیگنال کافی است متد `get_fft` را فراخوانی کنیم.

به علاوه هنگامی که مانند این پروژه، اطّلاعات سیگنال درون یک فایل متنی به صورت خط‌به‌خط قرار دارند، می‌توان به صورت مستقیم با فراخوانی `Signal.generate_from_file` یک نمونه از کلاس سیگنال را از روی آن فایل ساخت.

رادیو

وظیفه‌ی اصلی کلاس `Radio` استخراج سیگنال پیام از داده‌های دریافتی توسّط آنتن است. هنگام ساخت رادیو، ما صرفاً درصد خطای قابل تحمّل و سیگنال دریافتی را به عنوان ورودی به سازنده می‌دهیم.

سپس می‌توانیم هر بار هر فرکانسی را که مدنظر داریم از طریق فراخوانی متد `demodulate` دریافت کنیم.

در این متد ابتدا دو فیلتر از طریق فراخوانی تابع `best_filter` گرفته می‌شوند. وظیفه‌ی فیلتر اوّل استخراج کانالی است که می‌خواهیم صدای آن را استخراج کنیم و وظیفه‌ی فیلتر دوم این است که فرکانس انتقال یافته را جدا کند.

ابتدا با فراخوانی متد `get_channel_filter` فیلتر اوّلی را می‌گیریم. سپس آن را به همراه فرکانس مدّنظر به متد `get_shifted_signal` می‌فرستیم. وظیفه‌ی این متد این است که ابتدا با استفاده از `channel_filter` کانالی که می‌خواهیم را از سیگنال اصلی جدا کند. سپس با فراخوانی `get_shifted_signal` سیگنال انتقال‌یافته را دریافت می‌کنیم.

برای انتقال سیگنال به مرکز دستگاه مختصات، کافی است که سیگنال را در *Carrier* ضرب کنیم.

در پایان سیگنال انتقال‌یافته را به فیلتر پایین‌گذر می‌دهیم و خروجی را با مقیاس یک‌دهم کاهش می‌دهیم.

نتیجه که سیگنال صوتی مدّنظر ما است به شکل یک شئ از نوع `Signal` برگردانده می‌شود.

اجرای برنامه

برای اجرای این برنامه ما به **Python3.7** و پکیج‌های ذکرشده در فایل **requirements.txt** نیاز داریم.

به علّت حجیم بودن داده‌های ورودی و محدودیّت حجم بارگذاری در سایت، داده‌ی ورودی همراه با پروژه ارسال نشده است. **حتماً پیش از اقدام برای اجرای برنامه، فایل ورودی را دریافت و در محل دلخواه استخراج کنید.**

برای سهولت در استفاده، یک command-line interface در نظر گرفته شده است.

شیوه‌ی کلّی فراخوانی و کار با آن به شکل زیر است:

```
python3.7 AMRadio.py -v PATH -f int -e float -o PATH [-p]

```

شرح هرکدام از آرگومان‌ها در جدول زیر آمده است:

آرگومان	عملکرد	آیا این آرگومان اجباری است؟
v-	آدرس دقیق یا نسبی فایل متنی حاوی داده‌های سیگنال	بلی
f-	فرکانس کانال موردنظر به کیلوهرتز	بلی
e-	مقدار قابل قبول خطا	بلی
o-	آدرس دقیق یا نسبی ذخیره‌ی فایل صوتی نتیجه. پسوند wav باید به آخر نام فایل اضافه شده باشد.	بلی
p-	در صورت استفاده، فایل صوتی پس از ذخیره پخش خواهد شد	خیر

برای مثال برای دریافت و پخش رادیو آوا از فایل `input.txt` می‌توان دستور زیر را اجرایی کرد:

```
python3 AMRadio.py -v Data/input.txt -f 96 -e 0.01 -o
out/farhang.wav -p

```

فراموش نکنید که هنگام دادن آدرس برای فایل خروجی، فایل نباید وجود داشته باشد، امّا دایرکتوری‌های مشخّص شده در آدرس باید وجود داشته باشند.

فرکانس شبکه‌های مختلف در جدول زیر آورده شده است:

نام شبکه	فرکانس بر حسب کیلوهرتز
آوا	۹۶
اقتصاد	۱۴۴
گفت و گو	۱۹۲
فرهنگ	۲۴۰