تکلیف سری چهارم درس DSP – بخش کامپیوتری

«آشنایی با pyaudio »

نویسنده: مسعود بابایی‌زاده

تاریخ: ١٦ اسفند ١٣٩٩

١- مقدمه

مبحث Real-Time DSP یا «پردازش سیگنال در زمان حقیقی (یا بلادرنگ)» یعنی سیگنال ورودی در حین ورود پردازش شود و به صورت آنلاین به خروجی ارسال شود. در این مبحث اینگونه نیست که ما سیگنال ورودی را از قبل به صورت یک فایل ذخیره داشته باشیم و روی آن به صورت آفلاین پردازش انجام بدهیم و نتیجه را در یک فایل خروجی ذخیره کنیم. و چون پردازش باید به صورت Real-time انجام شود، نسبت به حالت آفلاین محدودیتهایی خواهیم داشت، مثلا:

الف) سیستم نمی‌تواند کاملا غیرعلی باشد. البته اگر بتواند با کمی تاخیر علی شود، مشکلی ندارد. در اینصورت خروجی همواره به همان مقدار نسبت به ورودی عقب است. که اگر این تاخیر کم باشد، ممکن است در کاربرد مورد نظر ما قابل قبول باشد.

ب) زمان مورد نیاز برای پردازش یک نمونه (ناشی از حجم محاسبات)، نمی‌تواند از زمان نمونه‌برداری بیشتر باشد. در غیر اینصورت قبل از اینکه فرصت کنیم یک نمونه را پردازش و نمونه متناظر خروجی تولید کنیم و به خروجی بفرستیم، نمونه ورودی بعدی از راه رسیده است.

ج) اگر ورودی و خروجی آنالوگ باشند، باید حواسمان به تعداد بیت A/D و D/A (کوانتیزاسیون) باشد. در پردازش آفلاین ما مشکلی نداریم که همه محاسبات را به صورت float یا double انجام بدهیم و در فایل خروجی ذخیره کنیم (یعنی در آنجا زیاد مشکل کوانتیزاسیون نداریم). ولی در اینجا، مثلا اگر D/A خروجی ٨ بیتی باشد، هر عددی برای خروجی به دست بیاوریم (مثلا با محاسبات double) باید متوجه باشیم که در هر حالت عددی که به خروجی می‌فرستیم باید یک عدد ٨ بیتی باشد. به خصوص حواسمان باید به رنج دینامیکی خروجی باشد (به عبارتی به overflow و underflow).

در این درس، احتمالا چند تکلیف کامپیوتری برای پردازش real-time خواهیم داشت. که در آنها مثلا ورودی، میکروفون کامپیوتر، و خروجی، بلندگوی کامپیوتر است. و قرار است صحبتی که در میکروفون می‌کنیم به صورت real-time پردازش شده و از خروجی بلندگو پخش شود.

در تکلیف حاضر، به عنوان مقدمه‌ای برای اینکار، هدف آشنایی با پکیج pyaudio در پایتون است، که به ما امکان کار با میکروفون و بلندگو (ورودی و خروجی صوتی) را در کامپیوتر، و به صورت real-time می‌دهد (همچنین برای خواندن و نوشتن در فایلهای صوتی، در کنار pyaudio می‌توان از پکیج wave استفاده کرد).

در این تکلیف قرار نیست پردازش خاصی پیاده‌سازی شود، بلکه فقط قرار است سیگنال میکروفون خوانده شود و به صورت real-time روی یک شکل رسم شود (یک ویدئو از این موضوع قرار داده‌ام؛ نگاه کنید). همچنین سیگنال میکروفون خوانده شود و بدون پردازش خاصی به بلندگوی کامپیوتر فرستاده شود.

۲- نکات اولیه در مورد pyaudio

قبل از اینکه به سراغ خواندن مستندات پکیج pyaudio بروید، چند نکته در مورد آن لازم است که بیان شود، تا آن مستندات واضح‌تر شوند:

- Pyaudio فقط یک اینترفیس پایتون است به کتابخانه PortAudio که کتابخانه‌ای است به زبان C و برای کارکردن با ورودی/خروجی صوتی کامپیوتر. PortAudio نسبتا قدیمی و جاافتاده است و استفاده از آن برای کارکردن با ورودی/خروجی کامپیوتر متداول است. Open-source هم هست و روی تمام سیستم‌عاملها هم هست (cross-platform). بنابراین توابع pyaudio در واقع توابع پایتونی هستند که کارشان در نهایت صدا زدن توابع متناظر از کتابخانه PortAudio است. به همین دلیل نگاهی به مستندات خود PortAudio می‌تواند مفید باشد (در آدرس http://www.portaudio.com/docs.html).
- به همین دلیل بالا، pyaudio از NumPy استفاده نمی‌کند، یعنی خروجی‌ای که از میکروفون می‌دهد و ورودی‌ای که برای ارسال به بلندگو دریافت می‌کند آرایه‌های NumPy نیستند، بلکه دنباله‌ای از بایتها هستند (بافری معمولی در C، که آرایه‌ای از بایتها است). این تایپ در زبان پایتون اسمش bytes است. **برای تبدیل bytes به یک آرایه numpy می‌توانید از تابع numpy.frombuffer استفاده کنید و برای تبدیل یک آرایه numpy به bytes می‌توانید از متد tobytes استفاده کنید** (یکی از متدهای هر آبجکت آرایه numpy است. یعنی مثلا اگر y یک آرایه numpy باشد، دستورش می‌شود y.tobytes). توجه شود که

frombuffer کل بافر ورودی خودش را در آرایه خروجی «کپی» نمی‌کند، بلکه فقط نوعی نگاه جدید به همان بافر تولید می‌کند (پس خیلی سریع است). به زبان numpy، آرایه خروجی مالک داده‌اش نیست (does not "own" its data) و آن داده قابل نوشتن هم نیست، یعنی نمی‌تواند در آن هم تغییری بدهد. یعنی مثلا اگر نوشتید:

```
x = np.frombuffer(data, dtype=np.int16)
```

در اینصورت x یک آرایه numpy است، ولی read-only است و نمی‌توان در داده‌های آن تغییری داد.

همچنین دقت شود که وقتی چند کانال داریم، در بافر اول یک sample از یک کانال است، بعد sample کانال بعدی از همان زمان، ... تا کانال آخر. بعد sample زمان بعدی از کانال اول، الی آخر (به عبارتی sample های کانالها interlace هستند). مثلا در حالت دو کانال چپ (L) و راست (R) سمپلها در بافر به صورت LRLRLRLR... هستند. همچنین اگر مثلا هر سمپل، int16 است (دو بایت) اینها به صورت Little Endian ذخیره می‌شوند (اول Least Significant Byte). که این در پایتون اسم تایپش می‌شود "<i2". در نتیجه اگر در بافری بنام data به تعداد nchannels کانال داشته باشیم، و هر سمپل هم دو بایت (۱۶ بیت) باشد، دستور زیر:

```
Sig = np.frombuffer(data, dtype='<i2').reshape(-1, nchannels)
```

یک ماتریس (آرایه دو بعدی numpy) به نام sig برمی‌گرداند که هر ستون آن، سیگنال یکی از کانالها است.

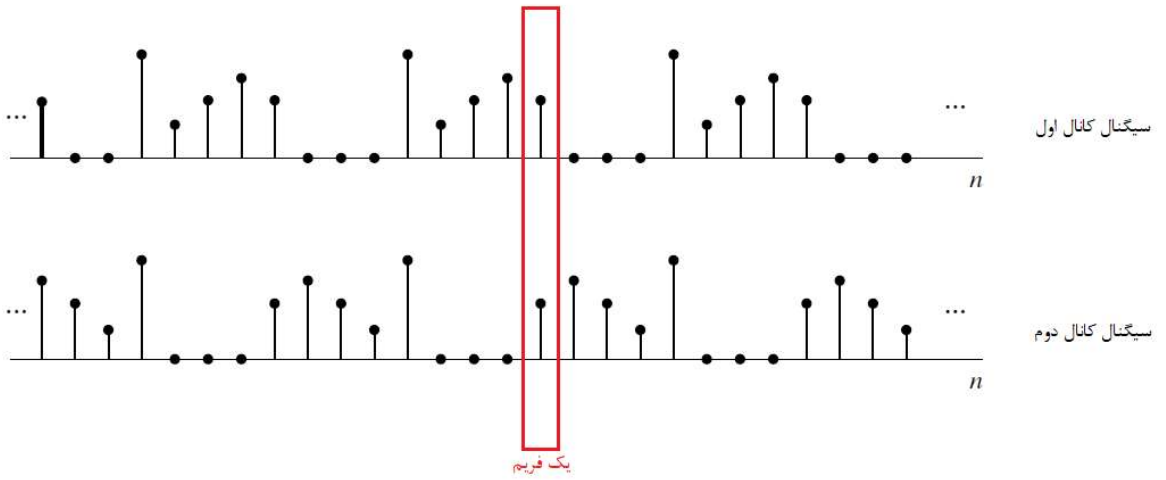
برای اطلاعات بیشتر در این مورد می‌توانید به لینک زیر بروید که اگرچه در مورد کارکردن با پکیج wave است، ولی از این نظر با pyaudio یکسان است:

https://nbviewer.jupyter.org/github/mgeier/python-audio/blob/master/audio-files/audio-files-with-wave.ipynb

توجه: یک اینترفیس جدیدتر به PortAudio که «پایتونی‌تر» باشد و مستقیما با آرایه‌های numpy کار کند، وجود دارد به نام python-sounddevice یا فقط sounddevice. ولی ظاهرا هنوز باگ دارد و در مرحله‌ای نیست که ما از آن استفاده کنیم. ضمن اینکه ظاهرا از pyaudio محدودتر است.

- مفهوم «فریم» (Frame) در pyaudio و wave و PortAudio: دقت کنید که در این پکیجها (هم در مستندات آنها و هم در اسم توابع آنها)، «فریم» معنی خاصی دارد. معمولا در پردازش سیگنال، یک فریم یعنی

(پنجره‌ای از) تعدادی نمونه متوالی یک سیگنال. **ولی در این پکیجها، «فریم» مطلقا به این معنی نیست**. ابتدا توجه کنیم که این پکیجها با چند سیگنال همزمان می‌توانند کار کنند. مثلا برای ارسال سیگنال استریو به بلندگو، ما در واقع دو سیگنال داریم که یکی به بلندگوی چپ می‌رود و دیگری به بلندگوی راست (این پکیجها بیشتر از دو سیگنال همزمان را هم پشتیبانی می‌کنند). در نتیجه کلمه «نمونه» (sample) در واقع یعنی «نمونه» از «یکی» از این سیگنالها. و در یک لحظه از زمان، ما از هر کدام از کانالها یک sample داریم که مجموعه این sampleها در این پکیجها «فریم» نام دارد. شکل زیر را ببینید:



با توجه به این معنای کلمه «فریم»، اگر w یک آبجکت wave باشد، آنگاه:

- تابع w.getframerate() چیزی که بر می‌گرداند، همان «sampling rate» درس ماست.
- تابع w.getnframes() تعداد «فریمهایی» را که در یک بافر (یا chunk) از داده هستند بر می‌گرداند.
- تابع w.getsampwidth() تعداد بایتهایی که هر sample (هر نمونه هر سیگنال) اشغال می‌کند را بر می‌گرداند.
- تابع w.getnchannels() تعداد کانالها را برمی‌گرداند.

پس اگر w.getnframes()=15 و w.getnchannels()=7 و w.getsampwidth()=2 (دو بایت یعنی ۱۶ بیت برای هر sample) باشد، انگاه اندازه بافر اشغالی حافظه بر حسب بایت می‌شود ۱۵×۷×۲ یعنی ۲۱۰ بایت.

- برای کار کردن با pyaudio ابتدا باید یک stream باز کرد (می‌تواند از نوع ورودی از میکروفون باشد، یا خروجی به بلندگو، یا هر دو). سپس از این استریم بلوکهایی (chunkهای) را خواند یا در آن نوشت. خواندن

یا نوشتن در این استریم می‌تواند به دو صورت مختلف انجام شود (که باید در هنگام باز کردن استریم مشخص شود): حالت Blocking و حالت Non-Blocking (یا Callback):

- **حالت Blocking**: در این حالت وقتی می‌خواهیم یک بلوک یا بافر (یک chunk) از استریم ورودی را بخوانیم (یعنی در واقع از میکروفون بخوانیم)، تابع stream.read را در pyaudio صدا می‌زنیم (که خودش در نهایت تابع Pa_ReadStream در PortAudio را صدا می‌زند). در اینصورت تا وقتی این بافر پر نشده باشد، برنامه از این تابع خارج نمی‌شود (طول بافر را کاربر هنگام بازکردن استریم با پارامتر frames_per_buffer تعیین می‌کند). یعنی فقط وقتی بافر پر شد، این تابع اجرایش تمام می‌شود و به برنامه ما برمی‌گردد و آدرس بافر پر شده را برمی‌گرداند.

توجه: پس از این مدت PortAudio کماکان دارد به خواندن میکروفون ادامه می‌دهد و در بافر دیگری (با همان اندازه‌ای که موقع بازکردن استریم تعیین شده است) در حافظه نمونه‌ها را ذخیره می‌کند. تا وقتی یک stream.read دیگر صدا زده شود و هر وقت آن قسمت حافظه که پر شد، آدرس بافر جدید را بر می‌گرداند. و اگر قبل از اینکه stream.read جدید صدا زده شود، این بافر جدید پر شود و PortAudio مجبور شود دیتایی دور بریزد، خطای PaInputOverflowed برمی‌گرداند و pyaudio هم وقتی این خطا را از PortAudio دریافت کند، خودش خطای InputOverflowed می‌دهد. که اینحالت ممکن است اگر اندازه chunk خیلی کوچک باشد، رخ بدهد.

برای سیگنالهای ارسالی به خروجی (استریمهای نوع خروجی) نیز مطالب مشابهی برقرار است (فقط در آنجا اندازه بافر خروجی موقع بازکردن استریم تعیین نمی‌شود).

- **حالت NonBlocking یا Callback**: در اینحالت خواندن و نوشتن در استریم به طور مستقیم انجام نمی‌شود. بلکه هنگام باز کردن استریم، یک تابع به عنوان Callback به استریم معرفی می‌شود (که می‌توانید آن را نوعی Interrupt Service Routine در نظر بگیرید). سپس هر وقت pyaudio نیاز به خواندن یا نوشتن یک بلوک جدید داشت (هر وقت کار بلوک قبلی تمام شد)، خودش این تابع را صدا می‌زند. به این ترتیب لزومی ندارد که ما در تابع خواندن از یک استریم، منتظر بمانیم و این زمان را می‌توانیم برای انجام پردازشهای دیگری استفاده کنیم.

ما در تکالیف این درس از این حالت دوم استفاده نخواهیم کرد. مشکل اصلی کار کردن با حالت Non-Blocking در پایتون این است که تابع Callback در یک thread جداگانه اجرا می‌شود. در نتیجه باید برنامه‌نویسی multi-thread را در پایتون بلد بود. مثلا در تکلیف فعلی به همین راحتی نمی‌شود matplotlib را در تابع Callback صدا زد. چون Matplotlib، Multi-thread safe نیست. پس باید تابع رسم شکل را در یک thread جداگانه گذاشت و هر ترد دیگری هم خواست چیزی رسم کند این تابع رسم را صدا بزند، و اگر توانست رسم می‌کند و اگر نه بگذارد در صف (یعنی هیچ ترد دیگری بجز این ترد رسم شکل، خودش اجازه نداشته باشد matplotlib را صدا بزند). و به اینصورت مشکلات race conditions حل شود. این روش پیچیده‌تر است و استفاده از آن هم چندان ضرورتی ندارد (دقت کنید که در همان حالت Blocking هم وقتی ما یک بلوک را خوانده‌ایم و داریم آن را پردازش می‌کنیم، در هر حالت PortAudio خودش دارد بلوک بعدی را می‌خواند و بافر دیگری را پر می‌کند. لذا ما هم داریم کار خودمان را انجام می‌دهیم. و تنها هنگامی که کار پردازشمان تمام شود، دستور خواندن یک بلوک جدید را می‌دهیم. لذا معطل شدن برای خواندن آنجا مشکل خاصی ایجاد نمی‌کند).

۳- صورت تکلیف

دو برنامه زیر را بنویسید:

مساله اول) برنامه‌ای که سیگنال میکروفون را بخواند و آن را مسقیما از بلندگو پخش کند، بدون هیچ پردازشی. در واقع در مستندات pyaudio برنامه اینکار با عنوان wire موجود است. ولی شما سه تغییر زیر را در آن بدهید، تا کدتان آماده شود برای انجام پردازشهای بعدی در تکالیف بعدی:

الف) بجای سیگنال استریو، با سیگنال مونو کار کنید.

ب) وقتی یک بافر ورودی را خواندید، آن را به یک آرایه numpy بنام x تبدیل کنید. سپس یک تابع signal_processing را صدا بزنید و x به به عنوان ورودی به آن بدهید، که این تابع هم یک آرایه numpy به نام y برمی‌گرداند (در این مساله این تکلیف، متن این تابع خیلی ساده فقط y=x.copy() است. یا حتی y=x هم فعلا کار می‌کند، چون تغییری نمی‌خواهیم بدهیم). سپس شما y را مجددا به bytes تبدیل می‌کنید و به بلندگو می‌فرستید.

موقع تبدیل y به bytes دقت کنید که تایپ خود y از نوع float نباشد و np.int16 باشد (اگر دارید ۱۶ بیتی کار می‌کنید). در واقع تابع signal_processing حتما باید خروجیش از نوع np.int16 باشد. حتی اینرا می‌توانید با دستور

```
assert y.dtype==np.int16
```

اجبار کنید (که بعدا یادتان نرود).

ج) بجای اینکه اینکار فقط چند ثانیه انجام شود، در همین وضعیت بماند تا کاربر Ctrl-C را فشار دهد. اینکار در پایتون اینطور انجام می‌شود که فشردن کلید Ctrl-C یک Exception بنام KeyboardInterrupt ایجاد می‌کند. که باید این Exception را Catch کنید، با استفاده از دستورات زیر:

```
try:
     while True:
          #your program of sending input to output
except KeyboardInterrupt:
     pass
# The rest of your program (mainly for clean exit, that is, stopping
# and closing the stream and terminating pyaudio).
```

در این تکلیف دقت کنید که هرچه اندازه chunk را زیاد کنید، delay بین خروجی و ورودی زیاد می‌شود، و احتمالا به دلیل برگشتن صدای بلندگو به میکروفون، اکو هم خواهید داشت. با زیاد شدن اندازه chunk، این اکو هم زیاد می‌شود. اگر اندازه chunk هم خیلی کوچک شود ممکن است InputOverflowed اتفاق بیفتد (شاید نه در حالت y=x که هیچ پردازشی ندارد، بلکه در حالت y=x.copy() این خطا را برای chunk خیلی کوچک ببینید).

در واقع شما حداقل به اندازه دو برابر زمان chunk، تاخیر خواهید داشت (یکی برای بافر ورودی و دیگری خروجی). البته این کل تاخیر نیست (چون بافرهای ورودی و خروجی احتمالی سخت‌افزاری و غیره هم هستند)، و تنها بخشی از تاخیر است که ما با تغییر اندازه chunk می‌توانیم آن را کنترل کنیم. در مورد تاخیر، می‌توانید این قسمت از مسندات خود PortAudio با ببینید:

http://www.portaudio.com/docs/latency.html

مساله دوم) سیگنال میکروفون را نمی‌خواهد به بلندگو بفرستید، بلکه با matplotlib آن را به صورت real-time رسم کنید (هر chunk را جداگانه رسم کنید). اینکار هم تا زمانی انجام شود که کاربر کلید Ctrl-C را بزند. یک ویدئو همراه این فایل تکلیف قرار داده‌ام که منظور را نشان می‌دهد، آن را نگاه کنید.

٤- چیزهایی که باید تحویل دهید

- یک برنامه پایتون (یک فایل با پسوند py) برای مسأله اول.
- یک برنامه پایتون (یک فایل با پسوند py) برای مسأله دوم.
- یک فایل ویدئو از دموی مساله دوم (مشابه فایلی که من برای شما گذاشته‌ام).

همه این موارد را به صورت یک فایل فشرده zip تحویل دهید.

٤- نکات و راهنمایی‌ها

۱) برنامه‌ها را به صورت نوت‌بوک ژوپیتر ننویسید، بلکه فایل پایتون معمولی باشد. به خصوص در مساله دوم، نوت‌بوک سرعت اجرای برنامه را پایین می‌آورد.

۲) از هر IDE استفاده می‌کنید، رسم شکل را به صورت inline انجام ندهید، بلکه در یک ویندو جداگانه شکل را رسم کنید. اگر از spyder استفاده می‌کنید، می‌توانید این نکته را اینجا تنظیم کنید:

Tools > preferences > IPython console > Graphics > Graphics backend > Backend: Automatic

سپس یا باید spyder را از اول اجرا کنید، یا اینکه کرنل IPython را restart کنید (با استفاده از دکمه تنظیمات، قسمت بالا سمت راست پنجره فرامین یا همان IPython console).

۳) در matplotlib اگر قرار باشد هر بار کل شکل از ابتدا رسم شود، زیادی کند خواهد بود. برای افزایش سرعت، فقط باید بخشی از شکل را که آپدیت می‌شود (خود قسمت منحنی، یا همان line در matplotlib) را رسم کنید. برای اطلاع از نحوه اینکار به آدرس زیر مراجعه کنید:

https://bastibe.de/2013-05-30-speeding-up-matplotlib.html

البته در لینک بالا، روش آخرش که سریعترین روش است، یک اشکال دارد و به شما خطای زیر را خواهد داد:

AttributeError: draw_artist can only be used after an initial draw which caches the renderer

برای اصلاح، باید همان کاری را که این پیام خطا به شما می‌گوید را انجام دهید، یعنی قبل از شروع حلقه، و قبل از دستور "`plt.show(block=False)`"، دستور "`fig.canvas.draw()`" را بگذارید. یعنی آن قسمت از کد چنین می‌شود:

```
fig, ax = plt.subplots()
line, = ax.plot(np.random.randn(100))
fig.canvas.draw()
plt.show(block=False)
ax.draw_artist(ax.patch)

tstart = time.time()
num_plots = 0
while time.time()-tstart < 5:
    line.set_ydata(np.random.randn(100))
    ax.draw_artist(ax.patch)
    ax.draw_artist(line)
    #fig.canvas.blit(ax.bbox)
    fig.canvas.update()
    fig.canvas.flush_events()
    num_plots += 1
print(num_plots/5)
```

قسمتهای قرمز رنگ تغییرات نسبت به کد موجود در مرجع بالا را نشان می‌دهند. دومی که با کامنت نشان داده‌ام (بجای سومین خط قرمز رنگ) این مزیت را دارد که با هر backend ای در matplotlib کار می‌کند. اگرچه در آن مرجع گفته انتخاب خوبی نیست، ولی در بعضی جاهای دیگر به همین دلیل آن را انتخاب خوبی دانسته‌اند.

توجه: اگرچه در این تکلیف ما برای رسم real-time سیگنال از matplotlib استفاده می‌کنیم (که با روش بالا برای افزایش سرعت آن برای ما کافی خواهد بود)، ولی راه‌حل اصولی‌تر استفاده از PyQtGraph است (http://pyqtgraph.org/)، که سرعتش خیلی بیشتر از matplotlib است.

٤) دو لینک زیر را در مورد pyaudio ببینید. اولی documentation کل نرم‌افزار است. در دومی، ٦ مثال می‌بینید که سه عمل record (خواندن از میکروفون و ذخیره در یک فایل wav) و play (خواندن یک فایل wav و پخش از بلندگو) و wire (اتصال مستقیم میکروفون به بلندگو) را هرکدام به دو روش (Blocking و Callback) انجام داده است، و این مثالها و توضیحاتی که در این فایل دادم، نقطه شروع خوبی برای شما خواهند بود:

https://people.csail.mit.edu/hubert/pyaudio/docs/
https://people.csail.mit.edu/hubert/pyaudio/

موفق باشید

مسعود بابایی‌زاده