KTH Datavetenskap
och kommunikation

Spektrala transformer
Laboration: DTMF-detektor

Introduktion

I denna laboration kommer du att bygga en detektor för att identifiera DTMF-signaler, dvs de tonsignaler som används för signallering i telenätet (tonvalstelefoner). Ditt program ska kunna ta in en ljudfil med DTMF-pulser (med godtycklig samplingshastighet), analysera den, och returnera korrekt knapp-sekvens som ASCII-tecken.

DTMF-signallering

En DTMF-signal består av två överlagrade sinustoner. På telefonens knappsats motsvarar den lägre frekvensen knappens rad och den högre frekvensen knappens kolumn, se tabell 1. Frekvenserna är ordnade i två icke-överlappande grupper, den låga och den höga gruppen, där den förra motsvarar raden och den senare kolumnen. Signalerna för A, B, C och D används sällan.

Vidare finns specifikationer för timingen hos DTMF-signaler: längden på varje ton måste vara minst 40 millisekunder, och mellan varje ton måste det vara minst 40 ms tystnad.

Utförande

Du ska utföra ett antal uppgifter med hjälp av Python, och svara på de frågor som ställs i peket. Laborationen utföres självständigt, antingen enskilt eller i grupp om två. Börja med att hämta `lab-dtmf-python.zip` från kurshemsidan. Denna fil innehåller några hjälpfunktioner och andra filer som du kommer att behöva. Filen`templates.py` innehåller platshållare/skelettkod för de funktioner du själv ska skriva.

Den kod som efterfrågas (och ska redovisas) ska oftast vara i form av python-funktioner. Ofta beskrivs även hur du kan testa att funktionen gör vad den ska. Du gör klokt i att skriva dessa tester i pythonfiler för att göra själva testandet rationellt och konsekvent.

Var noga med att dokumentera vad du gör och spara kod, plottar, bilder, ljudfiler och vad det kan vara till redovisningstillfället, då du ska redogöra för hur du kommit fram till dina resultat.

Tabell 1. Frekvenser för DTMF-signallering

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Uppgift 1: Manuell analys och segmentering

Börja med att titta på en DTMF-sekvens. Öppna filen `dtmf_all.wav` (den innehåller samtliga 16 signaler) i ett program för ljudanalys, t.ex. *WaveSurfer* eller *Audacity* och studera spektrogram och spektrum (*Spectrum section*.

Testa att ändra analysfönsterlängden (FFT-längd) och se vilken effekt det har på spektrum. Går det bra att separera de olika frekvnserna om man väljer ett väldgt kort analysfönster? Vad händer om man väljer ett långt analysfönster?

Passa även på att dela upp (segmentera) signalen i segment som motsvarar DTMF-tonpulser och spara varje puls i en egen wav-fil som du sedan sna läsa in och analysera i matlab.
Att redovisa: kommentarer

WaveSurfer-tips
I *Spectrum Section*-fönstret kan du jämföra två spektra genom att ta ett "snapshot" (med snapshot-menyn) och sedan flytta tidsmarkören.

Uppgift 2: FFT

Skriv ett pythonprogram som läser in en ljudfil och beräknar spektrum med hjälp av FFT, och plottar det med korrekt frekvensskala. Testa med olika fönsterlängder på samma sätt som i uppgift 1. Verifiera att resultatet stämmer med vad du fick fram i uppgift 1.
Att redovisa: .py-fil, spektrumplot

FFT, spektrum och frekvensskala
FFT är en algoritm för att beräkna den diskreta foruriertransformen *DFT* (dvs matematiskt är FFT och DFT samma sak). *FFT* tar som indata en array $x(n)$ av längden N ($x(n)$ är t.ex. sampel från en ljudsignal) och returnerar en array $X(k)$ av längden N, där index k motsvarar frekvens, dvs varje värde $X(k)$ säger i vilken grad frekvensen k ingår i signalen $x(n)$. Om $x(n)$ är en ljudsignal samplad med f_s så motsvarar k frekvensen kf_s/N. På grund av samplingsteoremet så behöver vi endast bry oss om frekvenser upp till $f_s/2$ dvs $k < N/2$.

I python (scipy) finns ett paket för beräkning av *FFT*, som importeras och används enligt nedan:

```
import scipy.fftpack as fft
Y = fft.fft(X)
```

Värt att notera är att Y är en *komplex* array. En vanlig användning av *FFT* är att beräkna *spektrum (effektspektrum)*. Detta fås genom att ta *FFT*:ns belopp i kvadrat, dvs i python `np.square(np.abs(fft.fft(X)))`

Uppgift 3: Avkodning

Nu ska du ta koden för beräkning av spektrum från uppgit 2 och utveckla den så att du kan detektera de åtta frekvenser som ingår i DTMF-standarden. Viktigt är att utnyttja det vi vet

om signalen: endast en frekvens i den låga resp. höga gruppen frekvensgrupen (se avsnittet om DTMF-signallering i inledningen) är aktiv vid ett givet tillfälle.

Följande steg behöver göras:

- hitta de värden på k som närmast motsvarar de sökta frekvenserna
- ta fram energin varje frekvens i den låga resp. höga gruppen.
- undersök samtliga 16 kombinationer för att hitta den mest troliga signalen. gruppen.

Antag att steg 2 ovan resulterar i två arrayer av längd 4, E_{low} och E_{high}, då skulle steg 3 utföras genom att multiplicera energierna i låga resp höga gruppen och jämföra med en fix tröskel:

```
if Elow(1)*Ehigh(1)) > T:
    str = '1'
elif Elow(1)*Ehigh(2)) > T:
    str = '2'
elif ...
```

Tröskeln kan bestämmas genom experimentering, men om koden ska vara robust bör man ta hänsyn till att absoluta energinivån kommer bero på både insignalens nivå och fönsterlängden, så det är lämplit att normalisera för dessa.

Skriv en funktion som gör ovanstående. Indata ska vara en array med sampel som motsvarar en DTMF-puls. Utdata ska vara en sträng med det tecken som kändes igen (bör vara tom sträng eller `' '` om inget tecken matchade!).

Att redovisa: python-funktion

Frivillig del

Uppgift 4: Sekvenser av pulser

Nu när du har koden som kan koda av en enskild DTMF-puls är en uppenbar vidareutveckling att göra det möjligt att tolka hela sekvenser av pulser. Det finns olika sätt att lösa detta, nedan skisseras två metoder, du kan välja själv vilken du vill göra.

Metod 1: semgentering med hjälp av energi

Denna metod bygger på att först dela upp signalen i enskilda pulser (det kallas för att segmentera signalen) och sedan anropa avkodaren för varje segment. Segmentering av DTMF-signaler är ganska tacksamt tack vare att specifikationen kräver tystnad mellan varje tonpuls.

Segmenteringen kan alltså göras helt baserat på signalens energi: varje gång signalenergin går från lågt till högt värde så börjar en ny puls, och när energin går ner igen så är pulsen slut (som ett mått på energi kan man ta signalen i kvadrat eller absolutbeloppet, det spelar ingen större roll här). För att kunna detektera pulser behöver vi ta ett *lokalt medelvärde* på energin, som sträcker sig över ett antal perioder av den lägsta DTMF-frekvensen. Detta lokala medelvärde får vi genom att lågpassfiltrera energisignalen, (kallas även för att "glätta ut" signalen). För utglättningen används exempelvis ett *rullande-medelvärde-filter* (moving average), vilket lättast görs med hjälp av matlabfunktionen `filter()`. För att välja lämplig längd på medelvärdesfönstret

kan du utnyttja vad vi vet från signalspecifikationen: alla giltiga DTMF-pulser är minst 40 ms långa och åtskiljda av minst 40 ms tystnad. Händelser kortare än denna tid är alltså ointressanta!

Pulserna kan sedan identifieras ur energikurvan med hjälp av tröskling, dvs när energin går över ett visst värde så har vi en puls, annars inte. För att detektionen inte ska vara beroende av insignalens absolutnivå, måste tröskelvärdet vara uttryckt relativt energikurvans toppvärde.

Skriv en funktion `dtmf_calc_energy()` som beräknar en glättad energikurva lämplig att segmentera signalen med enligt ovanstående princip. Inparametrar är en ljudvektor och längden på glättningsfönstret som ska användas. Notera att ett medelvärdesfilter av längd N fördröjer signalen med $N/2$, något som blir uppenbart om in- och utsignal plottas tillsammans. Funktionen ska kompensera för denna frödfröjning för att inte tappa "synk" i segmenteringen. Själva segmenteringen görs av den bifogade funktionen `dtmf_segment()`, som tar in energivektorn och returnerar en $k \times 2$-matris där varje rad innehåller start- och sluttid för ett segment, där k är antalet detekterade segment i signalen.

Testa din funktion genom att beräkna energin för filen `dtmf_all.wav`. Välj lämpligt värde på fönsterlängden, och plotta energikurvan tillsammans med vågformen.

Anropa sedan `dtmf_segment()` och kontrollera att segmenteringen stämmer: det ska bli 16 segment, ca 100 ms/800 sampel långa - jämför några start- och sluttider (som alltså returneras i sampelnummer) med vad du kan avläsa i *WaveSurfer*.

Slutligen ska du sätta ihop denna kod med avkodaren för att få ett program som klarar av att ta in en fil med flera pulser och skriva ut/returnera avkodad knappsekvens i form av en sträng.

Metod 2: blockbaserad analys

Denna metod är vanlig i många former av analys av ljud (tal, musik etc) och bygger på att dela upp signalen i block av fix längd (20 ms är lagom i detta fall). Varje block alyseras separat och kodas av till ett tecken (enl. uppg 3), och sedan sätts allt samman till en sekvens. Viktigt att notera i detta fall är att varje ton (som måste vara minst 40 ms lång enl standarden) kommer resultera i en rad av identiska tecken efter varandra, separerade av pauser (som alltså också måste gå att identifiera i avkodaren - de ska returnera en tom sträng!)

Exempel: antag att signalen innehåller sekvensen 1 - 2 - 3 där varje ton är 80 ms och pausen mellan tonerna är 40 ms. Då borde den blockbaserade analysen ge något i stil med `'1','1','1','1','','','2','2','2','2','','','3','3','3','3'`

Detta ska sedan kondenseras till `'1','2','3'`.

Filen `templates.py` innehåller skellettet till en funktion som kan användas för att gå igenom en längre array block för block på ovanstående sätt.

Zero-padding

Eftersom *FFT* ger tillbaka lika många punkter längs i frekvensdomänen som man har i insignalen i tidsdomänen (N) innebär det att korta sekvenser ger låg upplösning i frekvensdomänen. För att öka frekvensupplösningen kan man använda ett enkelt trick: fyll ut insignalen med nollor till önskad längd. Detta kallas *zero padding*. Exempel: vi vill analysera en sekvens $x(n), f_s = 8000Hz$ av längden $N = 160$ och önskar en frekvensuplösning på bättre än 15 Hz. En *FFT* av $x(n)$ skulle resultera i 160 frekvenspunkter fördelade från 0-8000Hz, dvs en frekvensuplösning på $8000/160 = 50.0Hz$ (se FFT, spektrum och frekvensskala ovan). Vi fyller ut $x(n)$ till fyra gånger den ursprungliga längden ($N = 640$, dvs 480 nollor) vilket ger en frekvensupplösning på $8000/640 = 12.5Hz$.

Att redovisa: python-fil