Matlab 4 – Fasvokoder

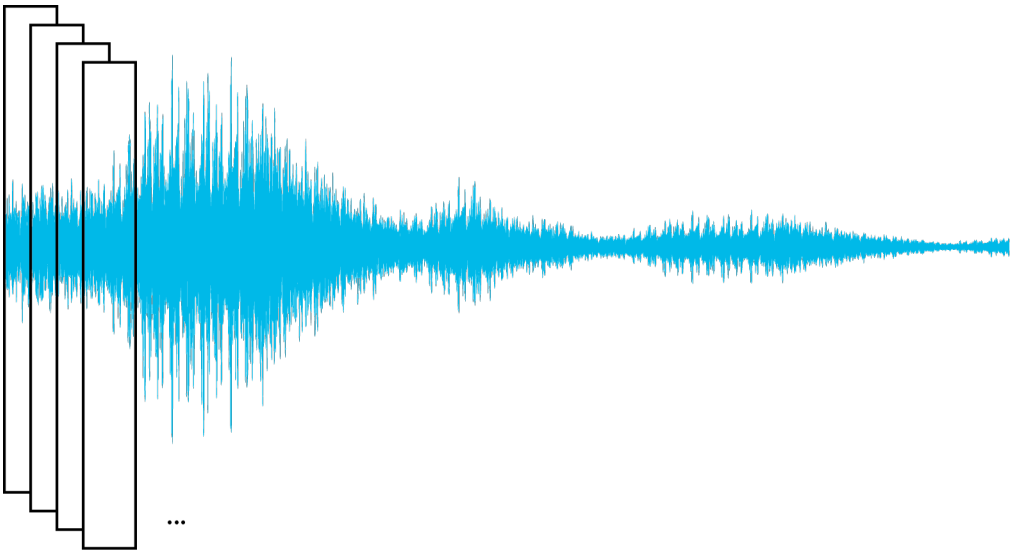
I denna laboration ska ni testa att göra time-stretch respektive pitch-shift i Matlab. Laborationen kommer att hållas på relativt enkel nivå för att testa grunderna och principen, men resultatet kommer inte att låta perfekt. Det gör inte så mycket, i stället kan vi diskutera grunderna i time-stretch/pitch-shift och fasvokodning, och utmaningarna där i.

Inför laborationen

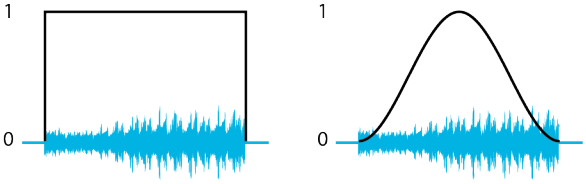
Ni ska med hjälp av Matlab skapa förkortning av en ljudfil och förlängning av densamma utan att påverka tonhöjden, ni ska också ändra tonhöjd utan att påverka uppspelningshastigheten. Läs därför först igenom denna text, och kolla länkarna.

Förberedelse – 1. Läs på om grunderna för fasvokodning

En fasvokoder delar upp ett ljud i ett antal fönster som överlappar varandra. Genom att flytta dessa fönster i förhållande till varandra vid uppspelning, kan tempot i ljudet förändras. Om fönstren spelas upp tätare kommer ljudet att gå fortare och spelas fönstren upp glesare kommer ljudet att gå saktare. Eftersom det inte är uppspelningshastigheten som ändras kommer ljudets originaltonhöjd/pitch att vara densamma.



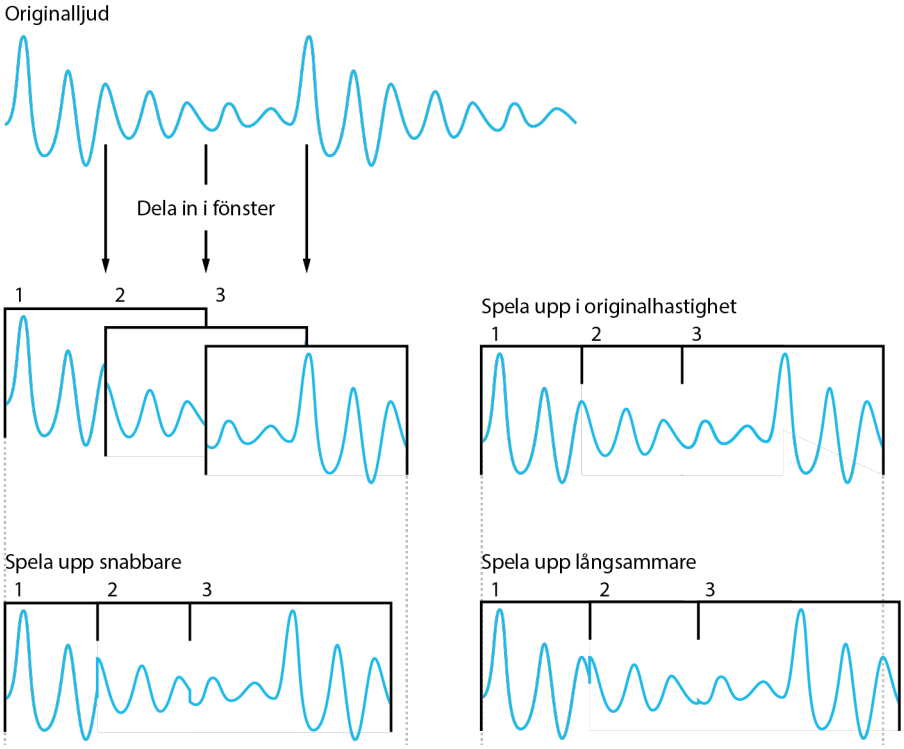
Detta är väldigt enkelt uttryckt och det är fler steg och aspekter som egentligen behövs tänkas på. Fönstren är normalt så kallade hanningfönster, dvs fönster som inte går rakt från 0 till 1, utan som har en mjuk övergång/övertoning mellan 0 och 1, respektive mellan 1 och 0. Detta påminner om `LFGauss` som man kan använda exempelvis som en envelop i SuperCollider. Det går att koda en sinus eller kosinus i Matlab och använda dessa för att skapa en olinjär fade. Skillnaden mellan Hann-, Hamming- eller Gaussian-fönster är formen på kurvan. Poängen är dock att skapa en fade, en mjukare övergång, mellan tyst och ljud.



Läs mer här:
https://en.wikipedia.org/wiki/Hann_function
https://en.wikipedia.org/wiki/Gaussian_function

https://doc.sccode.org/Classes/LFGauss.html
https://en.wikipedia.org/wiki/Window_function

Spelas ljudet upp i samma hastighet och i samma tonhöjd så är inte det något problem om ljudet klipps hårt från tyst till ljud (från 0 till 1), ljudklippen kommer att ligga i fas med varandra vid klippunkterna och uppspelningen sker smärtfritt och utan hörbara artefakter eller konstigheter. Men, problem kommer att uppstå när uppspelningshastigheten är snabbare eller långsammare än originalljudet eftersom vågformerna kommer att vara ur fas mot varandra och orsaka tvära förändringar (vågformen gör lodräta hopp) vilket i sin tur orsakar missljud och tillför frekvenser (hörbara exempelvis som knaster).



Det är så här denna laboration ska lösas, även om en riktig fasvokoder är mer avancerad och det är uppenbart att detta inte är en fullgod lösning av problemet. :-)
Läs mer här:
https://en.wikipedia.org/wiki/Phase_vocoder
https://sethares.engr.wisc.edu/vocoders/phasevocoder.html

I en riktig fasvokoder kommer varje fönster att gå igenom tre olika övergripande delar: analys, bearbetning och sammanställning (resynthesis). Hanningfönstren transformeras med FFT, Fast FourierTransform, och delas upp i bins, sinuskurvor, upp till halva samplingsfrekvensen (Nyquistteoremet). För varje bin får man fasinformation, frekvens och magnitud.
Läs mer här:
https://en.wikipedia.org/wiki/Fast_Fourier_transform
https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem

Därefter kan fasen bearbetas för varje bin så att fasen kan matchas mellan olika fönster, man adderar förflyttning av fasen tills den blir kontinuerlig.
Läs mer här:
https://en.wikipedia.org/wiki/Phase_(waves)

Till slut används en inversFFT för att återbygga ljudet.

Förberedelse – 2. Välj ett bra ljud att jobba med

Ni behöver ett bra ljud att arbeta med. Ljudet behöver ha någon form av takt så att ni hör när ni ändrar tempot, och det behöver även vara klingande, dvs ha toner så att ni hör när ni ändrar tonhöjd. Det räcker gott om ljudet är i mono, och har någorlunda bra samplingsfrekvens, men tänk på att högre samplingsfrekvens innebär mer data att beräkna vilket leder till längre väntetider under Matlabarbetet.

Ni får gärna spela in ett eget ljud, men tänk på att ni ska stå ut med ljudet medan ni jobbar...

Här finns en uppsjö olika ljud som ni kan ladda ned och använda:
https://freesound.org/
https://www.looperman.com/

Uppgiften

Uppgiften är att skapa en fulvokoder (i stället för fasvokoder) som klipper upp ett ljud i fönster och som sedan kan spela upp det i annan hastighet jämfört mot originalljudet (time-stretch), och i en annan tonhöjd (pitch-shift) än originalljudet, och oberoende av varandra.

Som vanligt finns olika sätt att lösa laborationen på, jag ger tips för en lösning, men detta är inte nödvändigtvis den bästa lösningen. Matlab har en del skumheter för sig, ”smarta” lösningar som är typiskt Matlabaktiga, men ni väljer att lösa uppgiften på det bästa sättet för er.

Läs in ljudfilen i Matlab

Det kan vara bra att inleda koden med att rensa allt som Matlab har igång och i minnet med

```
clearvars;
close all;
```

Fortsätt med att läsa in ljudfilen i Matlab. Jag brukar skriva min Matlabkod som en funktion, men ni väljer hur ni vill jobba. Kalla ljudvektorn för något bra, som exempelvis `originalSound` så att ni enkelt kan hitta tillbaka till originalljudet.

Ni kan överväga att göra ljudet till mono för att begränsa mängden beräkningar. Detta görs exempelvis genom att bara läsa ena kolumnen i `originalSound` och spara denna i `originalSound`, typ:

```
originalSound = originalSound(:,1);
```

Men, i den fortsatta koden kommer bara en kanal (kanal 1 i Matlab, vilket är vänster kanal) av ljudet användas oavsett om ni har gjort om ljudet till mono eller inte.

Sedan kan det vara bra att starta en `audioplayer` som kan spela upp originalljudet som ni kommer att lyssna till för referens. Kom ihåg att ange samplingsfrekvensen för originalljudet.

Med `figure` skapar ni ett nytt figurfönster i Matlab, och med `subplot(211)` följt av `plot(originalSound)` och `subplot(212)` följt av `plot(phaseVocodedSound)` kommer ni att kunna plotta de två vågformerna ovanför varandra. Om ni efter vardera plot lägger till `title('Original')` respektive `title('Phase vocoded')` så blir det ännu enklare att förstå det som visas.

För säkerhets skull, hör ni ljudet? Och kan ni se vågformen? Är ljudet bra valt för uppgiften, finns ton och tempo? Är ljudet lagom kort så att inte det går åt en massa beräkningstid i onödan?

_____

_____

_____

Specificera variabler för att skapa fönster av originalljudet

Skapa en ny variabel som ni ska använda för att enkelt påverka uppspelningshastigheten, jag valde att kalla den `speed`. Idén är att om `speed` är 1 så spelas ljudet upp i originalhastigheten, om `speed` är 0.5 blir det halva hastigheten och om `speed` är 2 blir det dubbla hastigheten.

Därefter ska fönsterstorleken bestämmas och stoppas in i en variabel (jag kallade den `windowSize`). Genom lite försök har jag bestämt mig för att 5% av samplingsfrekvensen är rätt lagom. Det ger en fönsterbredd på 2205 samples om samplingsfrekvensen är 44100Hz respektive ungefär 551 samples om samplingsfrekvensen är 11025Hz. Det kan vara bra, om än inte strikt nödvändigt, att avrunda det ni får ut.

En tredje variabel ska också skapas, och som anger stegstorleken. Om stegstorleken är större än fönsterstorleken kommer ljudet att spelas upp snabbare eftersom indexeringen av fönster ger utglesade överlapp, man skulle kunna säga att uppspelningen hoppar längre och kommer snabbare fram. Om stegstorleken är mindre än fönsterstorleken kommer indexeringen ge mer överlapp mellan fönstren, uppspelningen hoppar kortare och kommer att gå långsammare. Stegstorleken (`stepSize` i min kod) ska därför bli fönsterstorleken (`windowSize`) multiplicerat med hastighetsvariabeln (`speed`). Stegstorleken behöver vara ett heltal, så använd `round` även här. Går ni vilse i tankarna här så kolla tillbaka på bilden på sid 2.

Skapa en indexarray för det nya ljudet

Mycket av följande kod kommer från Stefan Gustavson som är en mycket större Matlab-guru än vad jag är. En del saker i den kommande texten är väldigt Matlab-specifik. Det går att lösa uppgiften på mer ”klassiskt” programmeringssätt, men då antagligen med mer kod.

Nu behövs en indexarray skapas för det nya ljudet. Med hjälp av denna indexarray kommer samples att läsas från `originalSound` enligt indexarrayens ordning och därigenom skapa det nya ljudet med rätt överlapp. Denna indexarray kommer alltså ha överlappande fönster när `stepSize` är mindre än `windowSize` så att uppspelningen går långsammare, respektive utglesade fönster när `stepSize` är större än `windowSize` så att uppspelningen går fortare.

Skapa en array med `zeros` som är tillräckligt lång med lite marginal. Multiplicera längden (`length`) av originalljudet `windowSize` dividerat med `stepSize`, och lägg sedan till `windowSize` som marginal. Gör arrayen endimensionell och avrunda för säkerhets skull längden på arrayen, för `zeros` vill ha heltal som argument. Så här ser det ut i min kod:

```
newFrame = zeros(round(length(originalSound) * (windowSize / stepSize) + windowSize), 1);
```

När indexarrayen är skapad, ska nollorna ersättas i den med sampleindex baserat på originalljudfilen men beroende på fönsterstorleken och stegstorleken. Detta görs med två for-loopar där den andra loopen sitter inuti den första. I dessa for-loopar har jag varit lat och använt `i`, `j`, respektive `k`. I detta fall tycker jag att det är ok med dåliga variabelnamn. :-P

Den yttre loopen ska gå från `0` i `stepSize` steg till längden (`length`) av originalljudet minus `1` (så att längden blir korrekt). Om man exempelvis skriver:

```
for i = (0:2:8)
    i
end
```

Kommer Matlab att skriva ut 0, 2, 4, 6, 8, alltså gå från 0 till 8 i steg om 2.

Den inre loopen ska gå från `0` till `windowSize` minus `1` (så att längden blir korrekt).

I den inre loopen ska indexarrayen (`newFrame` enligt min kod) fyllas på position `k` med värdet på den yttre loopen (`i` i min kod) plus värdet på den inre loopen (`j` i min kod) plus `1`. `k` som används för positionen i arrayindexen måste uppdateras varje gång i den inre loopen.

När indexarrayen är fylld med nya sampleindex kommer det med stor sannolikhet finnas kvar nollor från det att den skapades. Det är enkelt åtgärdat genom att skapa en ny array som innehåller booleans med 1 för värden större än 0 och 0 för alla värden som är 0. Exempelvis så här:

```
ix = newFrame > 0;
```

Därefter kan `ix` (enligt min kod) användas som ett index för indexarrayen, och det går att spara över indexarrayen i samma namn (`newFrame` enligt min kod):

```
newFrame = newFrame(ix);
```

Nu behövs det kontrolleras så att alla index i `newFrame` (enligt min kod) verkligen pekar på samples som finns inom originalljudets längd. Den här sortens "logic indexing" rätar dessutom ut matrisen till en enda kolumn, vilket är vad vi vill ha. På liknande sätt som när `newFrame` kontrollerades för att innehålla nollor ska `ix` (det är ok i det här läget att återanvända variabelnamnet) bli en array av booleans så länge de är kortare eller lika med längden (`length`) av originalljudvektorn (`originalSound` i min kod). Och, igen använder vi det nya indexet för att ersätta innehållet i indexarrayen på samma sätt som tidigare.

Därefter ska samples läsas från originalljudvektorn (`originalSound` i min kod) enligt indexarrayen (`newFrame` enligt min kod), och sparas i en ny variabel (`phaseVocodedSound` enligt mitt förslag i samband med plottande av vågformerna).

Spela upp det nya ljudet

Sätt `speed` till `1` och skapa en till `audioplayer` som kan spela upp det nya ljudet efter originalljudet. Plotta även den nya vågformen.

Hör ni någon skillnad mellan ljuden? Ser ni någon skillnad mellan ljuden?

_____

_____

_____

Prova därefter att sätta `speed` till `0.9`. Det ska göra att ljudet spelas upp i 90% hastighet jämfört mot originalljudet. Hur låter det? Prova också att ändra till `1.1` som är 110% hastighet. Funkar det? Hur låter det?

Nu kan det vara läge att testa att ändra på fönsterstorleken. Testa lite olika värden och lyssna, och fundera på hur ljudet förändras och varför.

Ändra tonhöjd på ljudet utan att ändra tempo

Nu ska koden användas för att i stället för att ändra tempo/uppspelningshastighet ändra tonhöjden (pitchen) på ljudet utan att påverka hastigheten.

Sätt först `speed` till `1`. Skapa sedan en ny variabel (`pitch` i min kod). Tanken med denna variabel är att 1 motsvarar originalets pitch, och 0.9 är 10% lägre pitch och 1.5 är 50% högre pitch.

Sedan ska `Fs2` skapas. `Fs2` är en ny samplingsfrekvens för det fasvokodade ljudet. Om uppspelningshastigheten sänks så att ljudet blir längre (men i bibehållen pitch) och sedan spelas ljudet upp i en proportionerligt högre samplingsfrekvens kommer tempot bli densamma som i originalet men pitchen kommer att vara högre. Om i stället ljudet görs kortare så att tempot går fortare, men spelas upp i en proportionerligt lägre samplingsfrekvens kommer ljudet att spelas upp i en lägre pitch men i originaltempot.

Skriv en `if`-sats. Om `pitch` är `1` ska den nya samplingsfrekvensen vara samma som originalljudets samplingsfrekvens. Men, om `pitch` är skiljt från `1` ska den nya samplingsfrekvensen bli originalets samplingsfrekvens multiplicerat med `pitch`.

Glöm inte att använda `Fs2` vid uppspelningen av det fasvokodade ljudet.

Hur låter det om `pitch` är `0.5`? Det är en halvering av frekvensen, dvs en oktav nedanför originalet. Hur låter det om `pitch` är större än `1`? Stämmer det att tempot är detsamma medan pitchen ändras?

Ändra både tempo och tonhöjd

Nu är all kod klar. Prova att sänka både tempo och pitch. Hur låter det? Testa att ha lägre pitch och snabbare tempo. Hur låter det? Osv… :-)

Fasvocoder vs granularsyntes

Att dela upp ett ljud i små beståndsdelar (grains) och sedan spela upp dem i olika ordning och med olika överlapp eller med upprepning av ljudfönstren kallas också granulärsyntes. Läs mer här:
https://en.wikipedia.org/wiki/Granular_synthesis

Läs in ett ljud, ta med samplingsfrekvensen, gör mono av ljudet, och normalisera upp ljudet till att vara mellan -1 och 1.

Därefter ska tre variabler deklareras, och dessa kommer sedan användas för att utforska granulärsyntesen:

- `grainSize` = storleken på varje grain i sekunder, och används när ljudet bryts ned i små fönster
- `speedFactor` = faktorn med vilken ljudets hastighet ändras, där mindre än 1 ger snabbare ljud och större än 1 långsammare ljud, och används när ljudet byggs ihop igen
- `numberRepeats` = antalet upprepningar varje grain ska ha och används när ljudet byggs ihop igen

Sätt `grainSize` till en 0.4 sekunder, `speedFactor` till 1 och `numberRepeats` till 1 så länge.

Skapa fönster

Först ska fönsterstorleken, i antal samples, skapas och stoppa in i en variabel (`frameSize` i mitt fall) genom att ta `grainSize` multiplicerat med samplingsfrekvensen avrundat nedåt.

Därefter ska det hopplängden för det uppdelade ljudet skapas och stoppas in i en variabel (`decomposeHopLength` i mitt fall) genom att ta `grainSize` multiplicerat med `frameSize` och sedan avrundat.

Nu kan antalet fönster (`numFrames`) beräknas genom att ta längden av originalljudet minus fönsterstorleken (`frameSize`) plus hopplängden (`decomposeHopLength`), och sedan dividera denna siffra med hopplängden. Detta bör sedan avrundas nedåt.

Baserat på detta kan en ny fönstermatris (`frameMatrix` i mitt exempel) skapas med funktionen `zeros` med en storlek av fönsterstorleken och antalet fönster (`zeros(frameSize, numFrames)`).

Dela upp ljudet i fönster

Skapa nu en for-loop, med en variabel som indikerar nuvarande fönster (`currentFrame`) som går från 1 (kom ihåg att Matlab börjar med position 1 i arrayer…) till längden av originalljudet (minus en fönsterstorlek) i steg om `decomposeHopLength`.

I loopen ska en variabel (`thisFrame`) tilldelas ljudet för det nuvarande fönstret. Gör det genom att ta nuvarande position (`currentFrame`) till nuvarande position plus fönsterstorleken (`frameSize`) minus 1 ur originalljudet.

Därefter kan fönstermatrisen (`frameMatrix`) bli tilldelad på position `:`, det avrundade värdet av nuvarande fönster delat hopplängden plus 1. Typ så här:

```
frameMatrix(:,round(currentFrame/decomposeHopLength)+1)
```

Ovanstående rad sätter positionen i fönstermatrisen och det värde som ska matas in är grainet (det korta ljudet) i nuvarande fönster (`thisFrame`) med ett Hannfönster.

Ett hannfönster skapar en envelop, liknande en gausskurva mellan 0 (helt dämpat ljud) och 1 (helt odämpat ljud), se mer här:
https://en.wikipedia.org/wiki/Hann_function

Detta görs i Matlab med funktionen `hanning`, läs mer här:
https://se.mathworks.com/help/signal/ref/hann.html

Och, hannfönstret ska sättas till fönsterstorleken och appliceras på det grainet (det korta ljudet i nuvarande fönster), vilket görs med `thisFrame .* hanning(frameSize).`

Återskapa ljudet

Det är nu dags att återskapa ljudet. Detta görs genom att skapa en variabel för den återskapandehopplängden (`reconstructHopLength`) genom att ta hopplängden som användes för att ta isär ljudet multiplicerat med variabeln med ljudets hastighet.

Sedan ska en array/matris (`reconstructedSound`) för det återskapade ljudet skapas med nollor (tänk på att det är ett monoljud), och längden ska vara originalljudets längd multiplicerat med antalet upprepningar (`numberRepeats`) och med `max` av 1 eller hastigheten (`speedFactor`).

Fundera på varför max av 1 eller hastigheten används. Vad händer om hastigheten ska vara snabbare än originalet om det då blir 1 här?

_____

_____

_____

För att sedan återskapa ljudet ska två for-loopar skapas, en yttre för nuvarande fönster (`currentFrame`) som ska gå från 1 till antalet fönster (`numFrames`), och en inre som ska gå från 1 till antalet upprepningar (`numberRepeats`). Det behövs också räknare (`countingFrames`) som deklareras till 1 utanför den yttre for-loopen och som räknas upp med ett för varje varv i den inre for-loopen.

Det är i den inre loopen som det nya ljudet sedan sätts ihop. Skapa först en startposition där det korta ljudklippet ska sättas in, baserat på den yttre räknaren (`countingFrames`) minus 1 multiplicerat med hopplängden (`reconstructHopLength`) för att återskapa ljudet plus 1.

```
startIndex = (countingFrames-1)*reconstructHopLength+1;
```

Därefter ska en stopposition skapas på samma sätt men med plus fönsterstorleken i stället för 1.

Start- och stoppositionerna ska anges som ett index (med start och stopp), och på detta index i det återskapade ljudet ska sedan ljudet på nuvarande fönster (`currentFrame`) i fönstermatrisen mixas med det ljud som eventuellt redan finns i det återskapade ljudet. Här behövs fönstermatrisen transponeras. Liknande:

```
reconstructedSound(originalSoundindex) =
reconstructedSound(originalSoundindex) + framematrix(:,currentframe)';
```

Nu har ljudet återskapats enligt storleken på grain, uppspelningshastighet och upprepning. Men, nu kan det återskapade ljudet behöva snyggas till i slutet och eventuella överskott av nollor tas bort. Ett sätt att ta bort nollor är med hjälp av `find`, liknande:

```
reconstructedSound = reconstructedSound(1:find(reconstructedSound, 1,
'last'));
```

Normalisera sedan det återskapade ljudet.

Lyssna på både originalljudet och det återskapade ljudet. Hörs det några skillnader, och i så fall, vilka är dessa och varför hörs de?

Ändra grainstorleken till 1. Blir skillnaden tydligare?

Ändra grainstorleken till 0.1 och hastigheten till 2. Vad händer med ljudet?

Testa sedan att ändra upprepningen till 4. Vad händer nu och hur låter det?

Till sist, använd 0.1 på grainstorlek, 0.5 uppspelningshastighet och 2 upprepningar. Vad händer nu och hur låter det?

Spela upp grains baklänges

En kul effekt med granulärsyntes är att spela upp varje enskilt grain baklänges men med en ordning som går i ”rätt” ordning från början till slut. I den inre for-loopen när det nya ljudet mixas (dvs `currentFrame` från `frameMatrix`) kan ni vända grainet med `fliplr`.

Hur låter det? Testa gärna olika inställningar och upprepningar.

Det går också att spela upp varje grain framlänges men med baklänges ordning. Ett sätt att göra detta på är att igen använda `fliplr` när det återskapade ljudet har normaliserats. Annars går det att skapa ett nytt tomt ljud (`reversedOrder`) och göra om de två for-looparna. När det återskapade ljudet sedan mixas hämtas position antalet fönster minus nuvarande plus 1. Liknande:

```
reversedOrder(originalSoundindex) = reversedOrder(originalSoundindex) +
frameMatrix(:,(numFrames-currentFrame)+1)';
```

Hur låter det? Testa gärna olika inställningar och upprepningar.

Spela upp grains i slumpad ordning

Det går också att spela upp ljud i slumpad ordning. Ett sätt att lösa detta på är att använda `randperm`, som är en funktion som skapar en slumpad permutation av något. Läs mer här:

https://se.mathworks.com/help/matlab/ref/randperm.html

Skapa en ny variabel och stoppa in den slumpade ordningen av `numFrames`.

Skapa ett nytt tomt ljud och två nya for-loopar, och när det återskapade ljudet sedan mixas hämtas position av den nuvarande ur det slumpade indexet. Typ:

```
randomSound(originalSoundindex) = randomSound (originalSoundindex) +
frameMatrix(:,randFrames(currentFrame))';
```

Hur låter det? Testa gärna olika inställningar och upprepningar.

Spela upp ljudet i annan pitch

Avslutningsvis går det att spela upp ljudet i en annan pitch, genom att kompensera samplingsfrekvensen vid uppspelning. Ett sätt att göra detta på är att skapa `Fs2` och ta samplingsfrekvensen (`Fs`) och multiplicera det med uppspelningshastigheten, och sedan använda `Fs2` vid uppspelning.

Hur låter det? Testa gärna olika inställningar och upprepningar.

Men, trots att denna version låter bättre (och kan göra konstigare saker med ljudet) så är det fortfarande inte en perfekt lösning av att två ljud går olika fort eller har olika pitch. Vilka problem kan uppstå när ljud mixas så här?

Vilka ljudförändringar uppstår när ljud med olika fas mixas ihop, och hur blir det hörbara? Hörs dessa problem i ljuden som används i denna laboration?