Beat detection

Ett projekt i kursen ”Signaler och System” ht02

Daniel Eklind, 781112 1479
Patrick Mansén, 700812 0292
Christian Melki, 771223 0031
Magdalena Psaros, 790710 0262

Sammanfattning

Taktidentifiering, eller “beat detection”, analyserar ett musikstycke och bestämmer den takt i vilken musiken spelas. Rapporten beskriver en algoritm för att utföra detta. Algoritmen utgår från en redan befintlig implementering i Matlab. Algoritmen har optimerats och förfinats och klarar i sitt nuvarande skick av att bestämma takter med stor noggrannhet och hög prestanda.

Innehållsförteckning

Inledning

Taktidentifiering är ett roligt och spännande verktyg i många sammanhang. Mest intressant är det kanske för musikindustrin och kommande projekt för att få datorer att synkronisera till musik i realtid. Detta kan vara i form av audiovisuella effekter och eventuellt musik komponerad med hjälp av datorer. Man kan visualisera virtuella miljöer med 3d-människor som dansar i takt till godtycklig form av musik. I vilket fall som helst är taktigenkänning ett grundläggande verktyg för att uppnå dessa mål.

Problemformulering

Den algoritm för taktidentifiering vi har valt att analysera och förbättra kommer ursprungligen från MIT Media Lab, och har implementerats i ett projekt vid Rice University. Vårt syfte är att grundläggande analysera vad denna algoritm gör, varför man har valt att implementera den på det sättet och hur man skulle kunna förbättra den. Genom att utföra flera, omfattande tester får vi en bättre och mer genomgripande förståelse av algoritmen och dess brister och kan med denna analys som utgångspunkt slutligen försöka förbättra och optimera algoritmen.

Bakgrund

Att hitta takten, eller bpm (beats per minute) i en låt kan teoretiskt göras på ett relativt enkelt sätt. Genom att utgå från låtens amplitudförändringar, antar man en takt och beräknar sedan det totala energiinnehållet över alla frekvenser i musiken. Därefter jämför man dessa energier och låter den med högst energi motsvara den takt man letar efter.
Algoritmen kan sägas vara uppdelad i fyra delmoment, där det första momentet transformerar signalen i frekvendomänen och delar upp den i ett antal frekvensintervall. Då man är intresserad av den totala amplitudförändringen inom varje frekvensintervall snarare än signalens frekvensinnehåll, omformar man sedan alla delsignaler genom att helvågslikrikta och falta dessa med ett hanningfönster i tidsdomänen. Därefter undersöker man signalernas derivata för att tydligare se när amplitudförändringar sker och halvvågslikriktar dem så att endast positiv derivata, d.v.s. amplitudökningar visas. Slutligen låter man signalerna passera en kamfilterbank där de taktfrekvenser som ska undersökas avgör avståndet mellan kammarna. Signalerna faltas med filtret för att man lättare ska kunna se vilka takter som innehåller mest energi. Den högsta energin kommer att motsvara takten i signalen.

Uppdelning i frekvensintervall

Signalen delas upp i sex olika frekvensintervall, för att på något sätt gruppera instrument som kan innehålla olika takter och som därför är lämpligast att analyseras var för sig. Intervallen kan bestämmas olika men har här valts så att varje frekvensband kommer att omfatta en oktav, enligt följande: 0-200 Hz, 200-400 Hz, 400-800 Hz, 1600-3200 Hz samt 3200 Hz till samplingsfrekvensen. Samplingsfrekvensen motsvarar därför den maximala frekvensen som är av intresse. Uppdelningen sker genom att man först fouriertransformerar signalen och får denna i frekvensdomänen. En vektor skapas sedan där de olika frekvensbanden placeras och där övriga frekvensområden, som ej är av intresse, sätts till noll.

Reducering och utjämning

För att underlätta analysen av frekvensinformationen vill man reducera denna till en nivå då man bara undersöker hur ljudet förändras i amplitud, d.v.s. få fram signalens amplitudenvelopp. Detta görs genom att man först helvågslikriktar signalen i tidsdomänen så att alla negativa amplituder vänds och blir positiva, vilket underlättar vidare bearbetning då man bara får positiva ljudamplituder att arbeta med. Därefter faltar man signalen med ett halvt hanningfönster för att få en mjukare och mer utjämnad kurva. Ett hanningfönster har samma form som en ökad (positiv) cosinuskurva. Denna har lågpasskaraktär med egenskapen att man lättare kan undersöka amplituderna även i sidloberna (en mer dämpad sidlobsutjämning till skillnad från ett rektangulärt fönster) på bekostnad av en bredare huvudlob.

Derivering och halvvågslikriktning

Eftersom människor känner igen takt på amplitudstegring så vill man modifiera amplitudenveloppen till att bara innehålla amplitudstegringar. Man anpassar helt enkelt innehållet till hur vi uppfattar takten. Detta görs enklast genom att börja inspektera alla sampel från början till slut. Om sampel N har lägre värde än N+1 så stiger givetvis funktionen och derivatan är därför positiv. Därför behåller man värdet i punkten N. Om derivatan hade varit negativ hade vi tagit bort värdet i punkten N. Eftersom man reducerar bort vissa sampel med avseende på derivatan så kan man säga att man halvvågslikriktar funktionen med avseende på just derivatan.

Kamfiltrering

Vid det sista steget är all data i optimal form för att hitta en eventuell takt. Så nu behöver man bara räkna ut den. Själva kamfiltersteget är i sig det mest beräkningskrävande steget i hela

analysen. Vår implementering spenderar någonstans mellan 80%-90% av tiden i denna sektion. Kamfiltret i sig är ingen komplex signal. Man kan sammanfatta ett kamfilter som en serie pulståg (diracpulser) med konstant period mellan pulserna. Själva filtreringen bygger på en väldigt enkel insikt om hur signaler beter sig när de multipliceras i frekvensdomän. Om kamfiltret med sina pulståg har samma periodtid som den halvvågslikriktade signalen ur föregående steg så kommer multiplikationen av de två signalerna ge ett svar som är förstärkt i energi eftersom pulstågen matchar de återkommande takterna med samma periodtid. Om inte periodtiderna matchar varandra så kommer inga förstärkningar att ske alls, eller bara en väldigt liten förstärkning p.g.a. brus och andra likartade fenomen. En förstärkning kan delvis ske på halvtakten 1/2*T. Även multipeltakter som 2*T och rationella takter som 3/4*T kan ge upphov till vissa förstärkningar. När vi har fått vår multiplikation av det fouriertransformerade filtret tillsammans med vår fouriertransformerade signal så summerar vi helt enkelt energiinnehållet i hela den serien och jämför den med tidigare högsta värde. Detta görs för alla uppdelade frekvensband och för alla takter vi letar i. Om vi hittar ett nytt högsta energivärde så behåller vi det för att jämföra med kommande värden och sätter den troliga takten till just den takt vi analyserade för. Detta beror helt enkelt på att den takt som innehåller mest energi är just den takten vi letar efter.

Simulering

Labbmetodiken är väldigt enkel. För att göra en bra samling mätvärden behöver man en representativ samling av musik. Allt ifrån svåravläst takt med väldigt flytande musik till jazz och rejäl basgång. Vi har valt bland annat en buggskiva med definerad takt på omslaget till cdn. Detta omslag använde vi då som referens till takten som algoritmen hittade.

Resultat

Under implementeringen och förfiningen av algoritmen från MIT så upptäckte vi ett antal saker. Vissa tillhör kategorier som direkt påverkar algoritmens funktionalitet och teori, andra kommer från implementeringsspecifika saker i Matlab. Vår inriktning blev väldigt mycket att förfina och trimma algoritmen i tid och i resultat. Implementeringen från Rice universitet gör en del mindre bra antaganden men fungerar i övrigt bra och ungefär som algoritmen från MIT är tänkt att fungera.

Inläsning av wav-filer

I implementeringen från Rice universitet upptäckte vi att de försökte läsa in hela wav-filer i minnet på datorn. Detta kunde leda till väldiga fördröjningar då en hel wav-fil på ungefär 40M skulle laddas in. Så vi förfinade inladdningsimplementeringen lite genom att endast läsa in de block i filen som vi kommer att analysera på. Denna enkla förändring gjorde att inladdning av stora filer gick flera faktorer snabbare. I Matlab har man nämligen möjlighet att specificera mellan vilka block inladdning ska ske, något som lämnades outnyttjat i den gamla implementeringen. Så vi hämtar först storleken på filen och räknar sedan ut det område vi vill beräkna takten på och hämtar in det i minnet. I den ursprungliga implementeringen så läser man endast in 2.2 sekunders data från wav-filen. Men eftersom våra optimeringar har gett oss en rejäl prestandavinst så har vi valt att satsa en stor del av vinsten på att läsa in ett större block och operera på det. I den nya implementeringen läser vi in 6.6 sekunder data för att få med ett längre block och därmed eventuellt bättre noggrannhet. En annan liten implementeringsfix är att koden numera klarar av att läsa stereofiler och tar då sedan den första av de två kanalerna utan att klaga.

Nedsampling av inläst data

Eftersom algoritmen är baserad på att huvudsakligen försöka hitta takt i de lägre segmenten av frekvensbanden så lämnas de övre i stort sett outnyttjade. Det brukar sällan finnas någon takt i frekvenser över 4kHz. Den tidigare implementeringen tog dock inte hänsyn till att de faktiskt bara behandlade de lägre segmenten utan tog in hela frekvensbandet och segmenterade det i delar. Det ledde till att det fanns en hel del icke taktgivande energi i signaler ovanför 4kHz. Detta har tagits till hänsyn i den nya implementeringen. Vid inläsning så läses samtidigt samplingsfrekvensen i wav-filen in för att senare behandlas av nedsamplingsalgoritmen. Nedsamplingen går helt enkelt till på det sättet att man slår ut en viss kvot av sampel i förhållandet mellan 8kHz till samplingsfrekvensen. T.ex. när man får in ett stycke med 44.1kHz samplingsfrekvens så behöver man bara behålla ungefär vart femte sampel i serien. Detta gör att antalet beräkningar som behövs för att beräkna takt blir mycket mindre och därmed blir givetvis algoritmen mycket snabbare.

Frekvensbandsuppdelning

Efter en del laborationer och analyser av algoritmen med ett antal olika sampel så hittade vi en möjlig miss i beskrivningen av frekvensuppdelningen. Både algoritmen och implementeringen föreskriver en ganska strikt frekvensuppdelning i oktaver men det sägs ingenstans om varför och hur frekvenskuppdelningen är vald. Vi gissar på att uppdelningen är vald med avseende på instrument, basfattig musik och hur den logaritmiska skalan för vår hörsel fungerar. I algoritmen föreskrivs en filtrering med tjebysjov-filter för att segmentera frekvenserna. Men implementeringen från Rice använder inte filter för segmenteringen utan bara rektangulära fönster och därmed fås en brant övergång mellan passband och spärrband. En riktig filtrering skulle nog leda till färre tidsbundna effekter p.g.a. rektangulära kapningar i frekvensdomän. I testerna har även ingått att ändra den lägsta gränsfrekvensen, från 200 Hz till 100, 150 och 220 Hz och samtidigt justerat frekvensintervallen så att de alltid är indelade i oktaver. Resultatet visade sig dock bli sämre. Vi har även analyserat vad som händer om man väljer att inte frekvensuppdela alls. Detta resultat blev faktiskt lika bra som den ursprungliga men med snabbare beräkningstid. Det förutsätter dock att beräkningen sker på en längre del av signalen för att få samma kvalitet. Men totalt sett så går det snabbare att inte frekvensuppdela medan man samtidigt beräknar över en längre tid. Tidsmässiga resultat för båda typerna av beräkningar finns att hämta i appendix.

Utjämning med Hanningfönster

För utjämning av amplitudenveloppen faltar man signalen med ett halvt hanningfönster. Enligt den ursprungliga dokumentationen säger man sig använda ett hanningfönster med den totala längden 0.4 sekunder men i själva implementeringen har vi upptäckt att man istället använder sig av längden 0.2 sekunder som dessutom halveras då man bara vill ha den andra hälften av fönstret. Denna längd har också visats vara en av de optimala enligt våra tester, större längd resulterar i fler antal misslyckade taktidentifieringar. Även längden 0.15 sekunder fungerar bra, då denna längd ger identiska resultat. Mindre längd (t.ex. 0.10) ger däremot en sämre precision. Det halva hanningfönstret är en liten mystisk historia i sig. Det finns inga referenser i algoritmspecifikationen till varför man väljer ett hanningfönster än mindre varför just halva. Det finns dock externa referenser på det halva hanningfönstret som pekar på ”temporal integration” som härmar örats egna sätt att uppfatta ljud. Detta har effekten att man tonar alldeles för snabba amplitudförändringar och framtonar en mjukare amlitudenvelopp istället.

Kamfiltrering

Kamfiltret är den överlägset mest beräkningsintensiva delen av algoritmen och även här har den nya implementeringen fått sig en del optimeringar. Bland annat har vi tittat på upplösningen av kamfiltreringen och hur den går till. Den ursprungliga implementeringen börjar med att stega 2 BPM steg genom hela BPM-rymden från 60 till 240. Sedan fortsätter den gamla implementeringen med att söka igenom det tidigare hittade max på 0.5 BPM, 0.1 och 0.01 BPM. Detta leder till en mängd onödiga beräkningar eftersom de flesta musikstycken är definerade på som bäst ett heltal när. Det är inte innan man ska mixa musik i realtid man behöver den extrema upplösningen. Dessutom finns det en dålig sidoeffekt av att börja med att gå igenom hela rymden på t.ex. alla jämna BPM. Om ett musikstycke har ett udda BPM är det risk för att den här metoden missar toppen på det udda BPM:et och väljer en halvtakt eller en rationell takt istället för den riktiga. Den nya implementeringen väljer helt enkelt att begränsa sig på att söka heltals-BPM och gör så genom att stega hela rymden sekventiellt, med både udda som jämna BPM. Effekten blir något färre missar och en initialt högre beräkningskostnad. Men eftersom vi hoppar över alla beräkningar för sub-BPM precision så lönar detta sig i längden.

En annan intressant modifikation av orginalimplementeringen är hur specifikationen av vissa operationer har gått till. I algoritmbeskrivningen så tar man och multiplicerar fouriertransformen av kamfiltret med fouriertransformen av signalen i frekvensbandet. Sen tar

man absolutbeloppet av varje nytt element och kvadrerar det för att få energin. Men vad är nu detta för operation då? Om man tänker efter så är ju absolutbeloppet summan av kvadraterna på real och imaginärdelarna i det komplexa talet varpå man drar kvadratroten ur det nya uttrycket för att sedan kvadrera det igen. Poängen är att det är onödigt att dra kvadratroten ur något som man senare ska kvadrera. Så modifikationen blev helt enkelt att bara multiplicera med komplexkonjugatet vilket leder till samma effekt som tidigare. Med denna förenkling har vi sparat 50 % beräkningskapacitet. Istället för två kvadreringar, en summa, en kvadratrot och ytterligare en kvadrering har algoritmen endast kvar de två kvadreringarna och summan, en högst anmärkningsvärd vinst har vi alltså gjort här.

Att taktigenkänning är beräkningsintensivt råder det vid det här laget inga tvivel om. Men var alla beräkningar åtgår var lite svårare att se innan man började använda matlabs inbyggda profilerare. ”Profiler” som den heter är ett utmärkt verktyg för att gå igenom sin kod med massor av testexempel. Det visade sig väldigt snart att det var i just den optimerade delen av kamfiltret koden spenderade ungefär 80 % av tiden i. Så det blev vitalt att segmentera och titta på hur koden fungerade i just de styckena för att se om eventuella optimeringar kunde göras. Hälften av vinsten från den nya, optimerade koden kommer ifrån användadet av den inbyggda profileraren.

Totalt sett så är våra beräkningsvinster minst 20-faldiga för godtycklig data. Hade syftet med den nya implementeringen blivit enbart optimeringsmässiga hade vi nog nöjt oss med detta. Men när man har sparat så mycket beräkningar kan man lika gärna lägga lite fler beräkningar på att öka kvaliten på utresultatet från algoritmen. Ett av de är givetvis att öka mängden data in till algoritmen. Men en annan är att öka antalet kammar som vi använder i kamfiltreringen. Ett kamfilters fouriertransform blir bara riktigt bra om kamfiltret sträcker sig i oändligheten åt båda hållen. Detta är givetvis högst teoretiskt och aldrig implementerbart om man ska göra en fouriertransform av ett oändligt filter. Men kvalitén på fouriertransformen blir givetvis bättre om vi har med fler kammar när vi transformerar. Orginalimplementeringen hade 3 kammar för 2.2 sekunder data. Vår implementering på 6.6 sekunder använder sig av 7 kammar och ger ett mycket bättre resultat än tidigare. Detta sker fortfarande på mindre beräkningsåtgång än tidigare.

Efter att ha prövat vår generella algoritm på en mängd olika musikstycken kan vi med säkerhet säga att det i de flesta fall finns rytmiskt återkommande amplitudstegringar i ljudet. Något som resulterar i mycket större energier för vissa kamfiltreringar än andra.

Vi kan även lägga märke till energitoppar utöver de som är allra högst, energitoppar som många gånger ligger besvärande nära i energimängd. Dessa utgör en källa till feltolkningar då

de många gånger ligger så nära att det kan tänkas var den lägre av de två som motsvarar den riktiga rytmen och att den andra, bara genom en slump, har tagit sig över den riktiga toppen.

Tidsdomän istället för frekvensdomän

En tidig idé för att förbättra algoritmen både i kvalitet och i hastighet innebar att helt enkelt utföra kamfiltreringen i tidsdomän. Vi har gått igenom kamfiltreringen och det borde stå klart att samma resultat skulle uppnås om tidssignalen helt enkelt adderades till sig själv, förskjuten med den sökta periodtiden (att notera är att den ursprungliga algoritmen från MIT använder sig av denna metod).

För att testa detta paddades signalen med nollor i båda ändar, tillräckligt många för att samma antal additioner som kamfiltret motsvarade skulle kunna utföras. Därefter utfördes de motsvarande additionerna varpå energin mättes.

Resultaten skilde sig avsevärt ifrån de som givits med kamfilter-metoden och då det visade sig att kamfilter-metoden gav korrekta resultat i många fall förkastade vi metoden att beräkna rytmen med faltning.

Fortsatt arbete

Antal fönster

Ett alternativ till ett längre fönster är flera fönster, att analysera olika stycken i musiken. En förutsättning för hela algoritmen är en konstant takt, något som är väldigt vanligt i musik generellt. Att analysera flera delar har två fördelar.

Man borde få ett bättre medelvärde om vi väljer mer spridda delar av musiken som förmål för analys. De olika rytmerna som hittas borde stämma bättre överens med den faktiska.

Eftersom FFT är en N log N-operation så kommer det gå snabbare att göra många korta än att göra en lång (t.ex. är $3(N \log N) < 3N \log 3N$). Man får mer precision för prestandan.

Problemet är att implementera en vettig beslutsmekanism till de fall då de olika analyserna ger olika resultat. Medelvärde är uteslutet då det kan skilja väldigt mycket när algoritmen väljer en halvtakt. Taktvärden som är helt fel borde ignoreras och möjligheter borde finnas att välja rytmer som slagit igenom mycket i alla analyser. Eventuellt bör statistiska analyser användas.

Ideala filter

I ursprungsimplementeringen användes ideala passbandsfilter för uppdelning i frekvensband. Detta kan ha förstört signalen så pass mycket att resultatet blivit fel. Om så är fallet kan våra antaganden, om att uppdelning i frekvensband är onödigt, vara felaktiga. Hade frekvensuppdelningen skett med realiserbara filter hade kanske resultatet blivit mycket bättre när signalen frekvensbandsuppdelats än när den inte hade det.

Slutsats

Orginalalgoritmen från MIT erbjuder en hel del parametrar att försöka ändra på för att förbättra resultatet och öka prestandan. I de tester som har utförts så har vi ökat prestandan men bibehållit kvaliteten på resultatet. Genom att öka antalet fönster och fönsterlängder kan vi alltid byta prestanda mot precision i resultatet. Men det har även visat sig att genom ett intelligent val av parametrar så går det att kombinera båda på ett väldigt lyckat sätt.
Detta innebär också att varje prestandaförbättring kan ses som en precisionsökning i algoritmen beroende på vilket mål man eftersträvar.
Uppdelningen i frekvensband enligt testerna påverkar inte resultatet och genom att undvika denna uppdelning har vi förbättrat prestandan på algoritmen.
Vi har även sett att en nedsampling till 8kHz inte påverkar resultatet och genom nedsampling har vi åstakommit avsevärda prestandaökningar.
Energinivårerna för olika kamfiltreringarna får väldigt spetsiga toppvärden. Detta innebär i princip att vi inte kan iterera oss fram till de maximala energivärdena utan man får göra ett antagande att takter är heltal och sedan avsöka hela taktrymden en gång.

Referenser

Scheirer, E. D. Music-Listening Systems. PhD thesis. MIT Media Lab, 2000.
http://citeseer.nj.nec.com/scheirer00musiclistening.html

Kileen Cheng, Bobak Nazer, Jyoti Uppuluri, Ryan Verret. Beat This, A beat synchronization project. Rice University, 2001.
http://www.owlnet.rice.edu/~elec301/Projects01/beat_sync/

Svärdström Anders. Signaler och System. Studentlitteratur, 1999

Steven W. Smith. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing, 1997
http://www.dspguide.com/pdfbook.htm

Appendix

Testresultat

Testprotokoll

Tabellförklaring:
blankt = felet mindre än +/-2 bpm
h = beräknad bpm hälften av riktig bpm
bpm-värde = felaktigt, beräknat bpm-värde
gult = Ursprunglig konfiguration

Test med olika fönsterlängder

Låt nr	BPM enl. konvolut	BPM med fönster 0,4	BPM med fönster 0.2	BPM med fönster 0.15	BPM med fönster 0.1
1	130				
2	170	108			
3	160	h	h	h	h
4	154				
5	148				
6	138				
7	126				
8	154				
9	170	85	176	176	176
10	164	h	h	h	h
11	144	147	147	147	147
12	136	h			h
13	144				
14	180				
15	176				
16	148	154	154	154	154
17	142				
18	144				
19	160				
20	140				
Antal h		3	2	2	3
Antal övriga fel		4	3	4	3
Totalt antal fel		7	5	6	6

Test med olika gränsfrekvenser (alltid oktavindelning, första bandet angivet)

Låt nr	BPM enl. konvolut	BPM med 0-100 Hz	BPM med 0-150 Hz	BPM med 0-200 Hz	BPM med 0-220 Hz
1	130				
2	170				90
3	160	h	h	h	h
4	154	148	148		
5	148				
6	138				
7	126				
8	154				
9	170	85	177	176	185
10	164	h	h	h	h
11	144		147	147	
12	136				68
13	144		148		
14	180	h	h		
15	176	89			179
16	148	77	77	154	153
17	142		232		
18	144				
19	160				
20	140				
Antal h		3	3	2	2
Antal övriga fel		4	6	3	5
Totalt antal fel		7	9	5	7

Test med olika rektangulära fönster, dels med startvärde 2 (vilket har totalt fyra iterationer), dels med startvärde 1 (vilket har totalt en iteration)

	Startvärde	2	1	2	1	2	1
Låt nr	BPM enl. konvolut	BPM med 2.2 s	BPM med 2,2 s	BPM med 4,4 s	BPM med 4,4 s	BPM med 6,6 s	BPM med 6,6 s
1	130						85
2	170				h	h	h
3	160	h	h	h	h	h	
4	154						
5	148						
6	138						
7	126						
8	154						
9	170	176		h	h	h	h

10	164	h	h	h	h		h
11	144	147					h
12	136						
13	144		155				h
14	180						
15	176				h		
16	148	154	153	152	152	152	152
17	142						
18	144			h	h	h	h
19	160						
20	140						
Antal h		2	2	4	6	4	6
Antal övriga fel		3	2	1	1	1	2
Totalt antal fel		5	4	5	7	5	8

Test utan frekvensuppdelning med en iteration med steg om ett BPM

	BPM enl. konvolut	BPM med 2.2 s	BPM med 4.4 s	BPM med 6.6 s
Låt nr				
1	130			
2	170			h
3	160	h		h
4	154			
5	148			
6	138			
7	126			
8	154			
9	170	176		
10	164	h		
11	144	147		h
12	136			
13	144			
14	180			
15	176		117	
16	148	154	152	152
17	142			
18	144			
19	160			
20	140			
Antal h		2	0	3
Antal övriga fel		3	2	1

Totalt antal fel	5	2	4

Kommentarer: Låt nu 16 har av resultaten att döma fel BPM-värde angivet på konvolutet.

Matlabkod

Fil bpm.m

```
function output=bpm(song, bandlimits)
% ----------------------------------xxxx----------------------------------
% ----------------------------------xxxx----------------------------------
%
% BPM takes in the name of one .wav file, and outputs its BPM
%
%    OUTPUT = BPM(SONG, BANDLIMITS) takes the
%    the name of one .wav file, as string, and output it's
%    bpm. BANDLIMITS and MAXFREQ are used to divide the signal for
%    beat-matching
%
%    Defaults are:
%       BANDLIMITS = [0 200 400 800 1600 3200]
% ----------------------------------xxxx----------------------------------
% ----------------------------------xxxx----------------------------------
%
% Set default bandlimits if not provided on the commandline
if nargin < 2, bandlimits = [0 200 400 800 1600 3200]; end

% Debug and print variables
printProgress = 0; % Set to 1 to print progress messages
printSongName = 0; % Set to 1 to print the name of the song
printSongInfo = 0; % Set to 1 to print info about the song/file
printTime = 0; % Set to 1 to print time consumption (the end total is always printed)
withPlots = 1; % Show plots
% ----------------------------------xxxx----------------------------------
% ----------------------------------xxxx----------------------------------
%
% Start timer. ( for debug purpose )
t1 = clock;
if printTime == 1
   timeconsumption = etime(clock,t1)
end

% Get file size
[SIZE samplfreq] = wavread(song, 'size');
songlength = SIZE(1,1);
maxfreq = floor(samplfreq/2);

% Debug
if printSongName == 1, song, end
if printSongInfo == 1, SIZE, samplfreq, songlength, maxfreq, end

% Length (in samples) of 6.6 seconds of the song and calculate
% start and stop position in our wavfile to be read.
% The choice of 6.6 seconds worth of sampletime is made after
% the lowest bpm-search made. At 60 bpm we get either 6 or 7
% beats within this interval. A minimum requirement to do
% useful calculations on.
sample_size = floor(6.6*samplfreq);
start = floor(songlength/2 - sample_size/2);
stop = floor(songlength/2 + sample_size/2);

% Debug and print time
if printSongInfo == 1, sample_size, start, stop, end
if printTime == 1, timeconsumption = etime(clock,t1), end
```

```
% Load song with calculated samplepositions.
if printProgress == 1, status = 'loading song...', end
sample = wavread(song,[start stop]);

% Keep the first column only (one channel)
% Preferred solution would be to make a mono-channel out
% of a stereo if possible.
sample = sample(:,1);
% ----------------------------------xxxx---------------------------------
% ----------------------------------xxxx---------------------------------
%
% DOWNSAMPLE PROCESSING
% Downsample if samplefreq >=16000 Hz.
% For speedup purposes. We do not need frequencies above
% 8KHz really. So we downsample by a factor of 2.
% So if the samplefrequency is above 16KHz it gets downgraded.
% Ie. 44.1KHz  -> 8.82KHz
if (samplfreq >= 16000)
   keepSample = floor(samplfreq/8000);
   newSample = zeros(floor(sample_size/keepSample), 1);

   for (i = 0:sample_size/keepSample-1)
      newSample(i+1, 1) = sample(keepSample*i+1, 1);
   end
   maxfreq = maxfreq/keepSample;
   samplfreq = samplfreq/keepSample;
else
   newSample = sample;
end

% Print debug
if printTime == 1, timeconsumption = etime(clock,t1), end

% Plot our wav-file after downsample.
% Plot the absolute value of the fft of our sample.
if withPlots == 1
   subplot(2,1,1); plot(sample);
   xlabel('Nr. of samples in time');
   ylabel('Normalized amplitude');
   title('Waveplot in time domain');
   subplot(2,1,2); plot(abs(fft(sample)));
   xlabel('Frequency');
   ylabel('Amplitude');
   title('Waveplot in frequency domain');
   pause
end
% ----------------------------------xxxx---------------------------------
% ----------------------------------xxxx---------------------------------
%
% FILTERBANK PROCESSING
% See filterbank.m for further comments.
if printProgress == 1, status = 'filtering song...', end
a = filterbank(newSample, bandlimits, maxfreq);

% Print debug
if printTime == 1, timeconsumption = etime(clock,t1), end

% Plot the the absolute values of the fft after segmentation in bands.
if withPlots == 1
   figure;
```

```
   subplot(3,2,1); plot(abs(a(:,1)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(1)),'Hz to ',num2str(bandlimits(2)),'Hz in frequency domain']);
   subplot(3,2,2); plot(abs(a(:,2)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(2)),'Hz to ',num2str(bandlimits(3)),'Hz in frequency domain']);
   subplot(3,2,3); plot(abs(a(:,3)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(3)),'Hz to ',num2str(bandlimits(4)),'Hz in frequency domain']);
   subplot(3,2,4); plot(abs(a(:,4)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(4)),'Hz to ',num2str(bandlimits(5)),'Hz in frequency domain']);
   subplot(3,2,5); plot(abs(a(:,5)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(5)),'Hz to ',num2str(bandlimits(6)),'Hz in frequency domain']);
   subplot(3,2,6); plot(abs(a(:,6)));
   ylabel('Amplitude');
   xlabel([num2str(bandlimits(6)),'Hz to ',num2str(maxfreq),'Hz in frequency domain']);
   pause
end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% HWINDOW PROCESSING
% See hwindow.m for further comments.
if printProgress == 1,  status = 'windowing song...', end
b = hwindow(a, 0.2, bandlimits, maxfreq);

%Print debug
if printTime == 1, timeconsumption = etime(clock,t1), end

% Plot the 6 bands of the windowed functions.
if withPlots == 1
   figure;
   subplot(3,2,1); plot(b(:,1));
   xlabel(['Samples ', num2str(bandlimits(1)),'Hz to ',num2str(bandlimits(2)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,2); plot(b(:,2));
   xlabel(['Samples ', num2str(bandlimits(2)),'Hz to ',num2str(bandlimits(3)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,3); plot(b(:,3));
   xlabel(['Samples ', num2str(bandlimits(3)),'Hz to ',num2str(bandlimits(4)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,4); plot(b(:,4));
   xlabel(['Samples ', num2str(bandlimits(4)),'Hz to ',num2str(bandlimits(5)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,5); plot(b(:,5));
   xlabel(['Samples ', num2str(bandlimits(5)),'Hz to ',num2str(bandlimits(6)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,6); plot(b(:,6));
   xlabel(['Samples ', num2str(bandlimits(6)),'Hz to ',num2str(maxfreq),'Hz in time domain']);
   ylabel('Amplitude');
   pause
end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% DIFFRECT PROCESSING
% See diffrect.m for further comments.
if printProgress == 1,  status = 'differentiating song...', end
```

```
c = diffrect(b, length(bandlimits));

% Print debug
if printTime == 1, timeconsumption = etime(clock,t1), end

% Plot the half-wave rectification of the six band.
if withPlots == 1
   figure;
   subplot(3,2,1); plot(c(:,1));
   xlabel(['Samples ', num2str(bandlimits(1)),'Hz to ',num2str(bandlimits(2)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,2); plot(c(:,2));
   xlabel(['Samples ', num2str(bandlimits(2)),'Hz to ',num2str(bandlimits(3)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,3); plot(c(:,3));
   xlabel(['Samples ', num2str(bandlimits(3)),'Hz to ',num2str(bandlimits(4)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,4); plot(c(:,4));
   xlabel(['Samples ', num2str(bandlimits(4)),'Hz to ',num2str(bandlimits(5)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,5); plot(c(:,5));
   xlabel(['Samples ', num2str(bandlimits(5)),'Hz to ',num2str(bandlimits(6)),'Hz in time domain']);
   ylabel('Amplitude');
   subplot(3,2,6); plot(c(:,6));
   xlabel(['Samples ', num2str(bandlimits(6)),'Hz to ',num2str(maxfreq),'Hz in time domain']);
   ylabel('Amplitude');
   pause
end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% TIMECOMB PROCESSING
% See timecomb.m for further comments
if printProgress == 1, status = 'comb filtering song...', end
e = timecomb(c, 1, 60, 240, bandlimits, maxfreq, printProgress, withPlots);

% Print debug
timeconsumption = etime(clock,t1)
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% OUTPUT RESULT
status = 'BPM of song is...'
status = e
```

```
function output = filterbank(sig, bandlimits, maxfreq)
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% FILTERBANK divides a time domain signal into individual frequency
% bands.
%
%    FREQBANDS = FILTERBANK(SIG, BANDLIMITS, MAXFREQ) takes in a
%    time domain signal stored in a column vector, and outputs a
%    vector of the signal in the frequency domain, with each
%    column representing a different band. BANDLIMITS is a vector
%    of one row in which each element represents the frequency
%    bounds of a band. The final band is bounded by the last
%    element of BANDLIMITS and  MAXFREQ.
%
%    Defaults are:
%       BANDLIMITS = [0 200 400 800 1600 3200]
%       MAXFREQ = 4096
%
%    This is the first step of the beat detection sequence.s
%
%    See also HWINDOW, DIFFRECT, and TIMECOMB
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Set default values if not provided on the commandline
if nargin < 2, bandlimits=[0 200 400 800 1600 3200]; end
if nargin < 3, maxfreq=4096; end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Transform our signal to frequency domain.
% This is our first transform in processing purpose.
dft = fft(sig);

% Get the length of the fft and of the number of bands.
n = length(dft);
nbands = length(bandlimits);

% Bring band scale from Hz to the points in our vectors
for i = 1:nbands-1
   bl(i) = floor(bandlimits(i)/maxfreq*n/2)+1;
   br(i) = floor(bandlimits(i+1)/maxfreq*n/2);
end

bl(nbands) = floor(bandlimits(nbands)/maxfreq*n/2)+1;
br(nbands) = floor(n/2);

% Create an output vector.
output = zeros(n,nbands);

% Create the frequency bands and put them in the vector output.
for i = 1:nbands
   output(bl(i):br(i),i) = dft(bl(i):br(i));
   output(n+1-br(i):n+1-bl(i),i) = dft(n+1-br(i):n+1-bl(i));
end

% Make sure the output vector at position 0,0 has value 0
% Because we dont want any DC-component in our signal!
output(1,1)=0;
```

```
function output = hwindow(sig, winlength, bandlimits, maxfreq)
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% HWINDOW rectifies a signal, then convolves it with a half Hanning
% window.
%
%    WINDOWED = HWINDOW(SIG, WINLENGTH, BANDLIMITS, MAXFREQ) takes
%    in a frequecy domain signal as a vector with each column
%    containing a different frequency band. It transforms these
%    into the time domain for rectification, and then back to the
%    frequency domain for multiplication of the FFT of the half
%    Hanning window (Convolution in time domain). The output is a
%    vector with each column holding the time domain signal of a
%    frequency band. BANDLIMITS is a vector of one row in which
%    each element represents the frequency bounds of a band. The
%    final band is bounded by the last element of BANDLIMITS and
%    MAXFREQ. WINLENGTH contains the length of the Hanning window,
%    in time.
%
%    Defaults are:
%       WINLENGTH = .4 seconds
%       BANDLIMITS = [0 200 400 800 1600 3200]
%       MAXFREQ = 4096
%
%    This is the second step of the beat detection sequence.
%
%    See also FILTERBANK, DIFFRECT, and TIMECOMB
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Set the defaultvalues if not provided on the commandline
if nargin < 2, winlength = .4; end
if nargin < 3, bandlimits = [0 200 400 800 1600 3200]; end
if nargin < 4, maxfreq = 4096; end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Get the length of the fft and of the number of bands.
n = length(sig);
nbands = length(bandlimits);

% Determine the length of the Hanning window to be used.
hannlen = winlength*2*maxfreq;

% Create a hanning window, fill it with zeros
hann = [zeros(n,1)];

% Create the right half of a Hanning window.
for a = 1:hannlen
   hann(a) = (cos(a*pi/hannlen/2)).^2;
end

% Take IFFT to transfrom to time domain.
for i = 1:nbands
   wave(:,i) = real(ifft(sig(:,i)));
end

% Full-wave rectification in the time domain.
% Ie. Take absolute value of the function.
```

```
% And then go back to frequency with FFT.
wave = abs(wave);
for i = 1:nbands
   freq(:,i) = fft(wave(:,i));
end

% Convolving with half-Hanning same as multiplying in
% frequency. Multiply half-Hanning FFT by signal FFT. Inverse
% transform to get output in the time domain.
for i = 1:nbands
   filtered(:,i) = freq(:,i).*fft(hann);
   output(:,i) = real(ifft(filtered(:,i)));
end


function output=diffrect(sig,nbands)
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% DIFFRECT differentiates a signal, then half-wave rectifies the
% result.
%
%    DIFF = DIFFRECT(SIG, NBANDS) takes in a time domain signal
%    stored in a vector with each column representing a different
%    frequency band. The number of frequency bands is passed in
%    through NBANDS.
%
%    Defaults are:
%       NBANDS = 6
%
%    This is the third step of the beat detection sequence.
%
%    See also FILTERBANK, HWINDOW, and TIMECOM B
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Set the defaultvalues if not provided on the commandline
if nargin <2, nbands=6; end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Get the length of the signal
n = length(sig);

% Create output matrix
output=zeros(n,nbands);

% Now loop through all the values in all the frequency-bands.
% We intend to keep values that have a positive derivative.
for i = 1:nbands
   for j = 5:n
      % Find the difference from one smaple to the next
      d = sig(j,i) - sig(j-1,i);
      if d > 0
         % Retain only if difference is positive (Half-Wave rectify)
         output(j,i)= d;
      end
   end
end
```

```
function output = timecomb(sig, acc, minbpm, maxbpm, bandlimits, maxfreq, printProgress, withPlots)
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% TIMECOMB finds the tempo of a musical signal, divided into
% frequency bands.
%
%    BPM = TIMECOMB(SIG, ACC, MINBPM, MAXBPM, BANDLIMITS, MAXFREQ,
PRINTPROGRESS, WITHPLOTS)
%    takes in a vector containing a signal, with each band stored
%    in a different column. BANDLIMITS is a vector of one row in
%    which each element represents the frequency bounds of a
%    band. The final band is bounded by the last element of
%    BANDLIMITS and MAXFREQ. The beat resolution is defined in
%    ACC, and the range of beats to test is  defined by MINBPM and
%     MAXBPM.
%
%    Defaults are:
%       ACC = 1
%       MINBPM = 60
%       MAXBPM = 240
%       BANDLIMITS = [0 200 400 800 1600 3200]
%       MAXFREQ = 4096
%       PRINTPROGRESS = 1
%      WITHPLOTS = 1
%
%    Note that timecomb can be recursively called with greater
%    accuracy and a smaller range to speed up computation.
%
%    This is the last step of the beat detection sequence.
%
%    See also FILTERBANK, HWINDOW, and DIFFRECT
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Set the defaultvalues if not provided on the commandline
if nargin < 2, acc = 1; end
if nargin < 3, minbpm = 60; end
if nargin < 4, maxbpm = 240; end
if nargin < 5, bandlimits = [0 200 400 800 1600 3200]; end
if nargin < 6, maxfreq = 4096; end
if nargin < 7, printProgress = 1; end
if nargin < 8, withPlots = 1; end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Get the length of the signal and of the number of bands.
n=length(sig);
nbands=length(bandlimits);

% Set the number of pulses in the comb filter
npulses = 7;

% For each band, transform the band to frequency domain. Get the FFT.
for i = 1:nbands
   dft(:,i)=fft(sig(:,i));
end

% Initialize values needed in the calculation of energy contents.
stepz=0;
```

```
maxe = 0;
energyplot = zeros(length(minbpm:acc:maxbpm),1);
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
for bpm = minbpm:acc:maxbpm
   % Initialize values needed in each turn of the loop.
   % Ie, set energy to zero, continue the stepping and create a new
   % filter.
   e = 0;
   stepz=stepz+1;
   fil=zeros(n,1);

   % Calculate the difference between peaks in the filter for a
   % certain tempo. nspace is the space between the combs in the
   % combfilter
   nstep = floor(120/bpm*maxfreq);

   % Print debug
   if printProgress == 1, percent_done  = 100*(bpm-minbpm)/(maxbpm-minbpm), end

   % Creation of the filter.
   % Set every nstep samples of the filter to one at nstep distance.
   for a = 0:npulses-1
      fil(a*nstep+1) = 1;
   end

   % Get the filter in the frequency domain.
   dftfil = fft(fil);

   % Calculate the energy after convolution
   % Ie. For all bands multiply the filter with the bank.
   % Then square the real and imaginary part. ie. abs.. but dont
   % take the square root, since the energy^2 will cancel the sqrt.
   % Also add upp the sum of the energy contained within the band.
   for i = 1:nbands
      x = dftfil.*dft(:,i);
      x = real(x).^2 + imag(x).^2;
      e = e + sum(x);
   end

   energyplot(stepz,1) = e;
   % If greater than all previous energies, set current bpm to the
   % bpm of the signal.
   if e > maxe
      sbpm = bpm;
      maxe = e;
   end
end
% ---------------------------------xxxx---------------------------------
% ---------------------------------xxxx---------------------------------
%
% Plot the energycurve of the signal. This is the energy contained in
% all bands of our sample. This should be a clear picture of where
% energy is contained.
if withPlots == 1
   figure;
   plot(energyplot);
   xlabel(['BPM from ',num2str(minbpm),' to ',num2str(maxbpm)]);
   ylabel('Energy');
```

```
    title('Energy content after comb filtration');
    pause
end

% Return the result.
output = sbpm;
```