Linköpings tekniska högskola
2004-03-17

Anders Fjeldstad
andfj645@student.liu.se

En enkel

ljudkompression

implementerad med

Java

- ett projekt i kursen TNG015 Signaler och system

Sammanfattning

Denna rapport beskriver ett mindre individuellt projekt i kursen TNG015 Signaler och system. Projektet gick ut på att utveckla ett Java-program för komprimering av digitalt ljud, där kompressionsgraden ökas genom att ljudet först lågpassfiltreras i frekvensplanet.

Ett experiment visar att det teoretiska resonemanget fungerar rimligt bra, och därmed är projektet lyckat. Själva programmet har vissa nackdelar som till exempel att det ibland producerar artefakter i den filtrerade ljudsignalen samt att det använder det fysiska arbetsminnet på ett mycket oekonomiskt sätt. Trots detta fyller det sitt syfte, vilket innebär att projektets mål är uppfyllt.

Innehållsförteckning

1 INLEDNING ... 3

1.1 BAKGRUND ... 3
1.2 SYFTE ... 3
1.3 METOD, KÄLLOR OCH KÄLLKRITIK ... 3
1.4 FÖRKUNSKAPER HOS LÄSAREN ... 4
1.5 TYPOGRAFISKA KONVENTIONER ... 4

2 GRUNDLÄGGANDE TEORI ... 4

2.1 PRINCIP FÖR EN ENKEL KOMPRESSION AV DIGITALA LJUDSIGNALER ... 4
2.2 WAVE-FORMATET I KORTHET ... 5
2.3 LJUDFILTRERING I FREKVENSPLANET ... 6

3 SAMMANFATTNING AV ARBETET ... 7

3.1 ARBETSGÅNG ... 7
3.2 PROGRAMMETS KLASSER – EN PRESENTATION ... 8
3.2.1 Sound_Data_Handler ... 8
3.2.2 Comp_GUI ... 8
3.2.3 FFT ... 8

4 EXPERIMENT ... 9

5 INSIKTER ... 10

5.1 MINNESEKONOMI ... 10
5.2 ARTEFAKTER I UTSIGNALEN ... 10

6 AVSLUTANDE ORD ... 11

7 REFERENSLISTA ... 11

BILAGA A: KÄLLKOD

Inledning

Med denna rapport vill jag redogöra för ett kortare projektarbete jag utfört enskilt inom ramarna för kursen TNG015 Signaler och system som ges vid Linköpings tekniska högskola på Campus Norrköping. Arbetet gick ut på att med hjälp av programmeringsspråket Java ta fram en enkel men principiellt fungerande ljudkompression för digitalt ljud.

1.1 Bakgrund

Efter att ha läst kursen Signaler och system fick jag möjlighet att göra ett extraarbete för högre betyg. Jag är intresserad av programmering och hade under flera laborationer insett att ljudfiltrering och komprimering verkade spännande – alltså valde jag att fördjupa mig litet inom detta område. Under kursen hade jag arbetat med liknande uppgifter med i programmet Matlab[1] som erbjuder många lättanvända hjälpmedel och såg därför en utmaning i att utvidga mina verktyg till ett ”rent” programspråk.

1.2 Syfte

Min ambition var att skapa ett Java-program som kunde läsa in en digital ljudfil i formatet Wave, filtrera bort frekvenser högre än en viss brytfrekvens, komprimera data enligt någon känd metod och sedan spara det komprimerade ljudet (tillsammans med formatinformation) till en ny fil.

Jag valde att koncentrera mig på själva frekvensanalysen och filtreringsprocessen, vilket är motiveringen till att jag inte själv försökt utveckla någon egen komprimeringsmetod eller FFT[2]. Initialt avsåg jag inte heller att ta fram något program för uppspelning av de komprimerade ljudfilerna utan lämnade detta öppet för genomförande i mån av tid. I slutändan gjorde jag en kompromiss i denna aspekt – programmet låter användaren lyssna på det filtrerade ljudet innan det komprimeras och sparas, men det finns ingen funktion för uppspelning av det komprimerade ljudformatet.

Kortfattat var projektets mål att utveckla en fungerande programvara som testar det teoretiska resonemanget kring filtrerings- och kompressionsprincipen.

1.3 Metod, källor och källkritik

De grundläggande kunskaperna för att genomföra projektet hade jag från kursen – det handlar egentligen inte om så mycket ny teori utan mer om att koppla ihop sådant jag redan kan. Efter en kort diskussion med kursens examinator kunde jag verifiera att min idé i huvudsak var korrekt men fick några tips om faktorer jag inte tänkt på tidigare.

Allmänt präglades mitt arbete av att läsa i kursboken av Kamen & Heck (2000) då jag behövde friska upp minnet gällande teorin, samt att söka programmeringstekniskt relaterad information i dokumentation och diskussionsforum på Internet. Vad gäller Internetkällorna så kan jag försäkra om att de är tillförlitliga – jag har helt enkelt märkt att det de beskriver fungerar i praktiken, vilket bör räcka för att lugna den tvivlande.

---

[1] Programpaket för diverse matematiska simuleringar och beräkningar

[2] Eng. *Fast Fourier Transform* – algoritm som implementerar diskret Fouriertransform för beräkning med dator

1.4 Förkunskaper hos läsaren

Jag kommer i denna rapport förutsätta att läsaren känner till grunderna inom signalbehandling motsvarande de kunskaper som kan insamlas till exempel under en kurs motsvarande TNG015 Signaler och system. Dock kommer jag att förklara vissa tekniska begrepp när jag tycker att det behövs för att intresserade personer utan dessa förkunskaper ändå ska kunna följa resonemanget hjälpligt.

Vidare är det nödvändigt att läsaren är bekant med grundläggande objektorienterad programmering i Java (eller motsvarande programspråk). Även här har jag valt att närmare förklara vissa termer (såsom programspråkspecifika klasser och metoder) för att underlätta för läsaren.

1.5 Typografiska konventioner

I fortsättningen kommer jag att skriva ”kursen” när jag syftar på TNG015 Signaler och system.

2 Grundläggande teori

För att genomföra uppgiften var jag tvungen att koppla samman kunskaper från kursen med tidigare erfarenheter inom programmering, samt söka efter information inom de områden som projektet omfattade men som jag inte kände till något om sedan innan. Nedan beskriver jag den teoretiska princip som ligger till grund för mitt arbete, tillsammans med mer specifika områden.

2.1 Princip för en enkel kompression av digitala ljudsignaler

En digital ljudsignal är en följd av sampelvärden som beskriver originalsignalens amplitudvärden. Samplingsfrekvensen avgör vilket frekvensområde som signalen kan omfatta, men vilka frekvenser som verkligen ingår i frekvensen är förstås helt beroende på ljudets natur och varierar från fall till fall. Allmänt kan man säga att höga frekvenser bidrar till detaljerade variationer i signalens amplitudvärden. Dessutom är det fullt möjligt för en ljudsignal att innehålla frekvenser som är svåra eller omöjliga att uppfatta för en människas hörsel.

När man komprimerar data avser man att minska det lagringsutrymme som upptas medan man bibehåller innehållet (eller det huvudsakliga innehållet). Det finns två typer av kompressioner, nämligen förstörande och oförstörande. En förstörande kompression tar bort en viss mängd information genom en icke-trivial princip som gör att man vid rimliga kompressionsgrader inte märker så stor skillnad. Detta är praktiskt tillämpbart på till exempel bilder, ljud och (naturligtvis) video, där det rör sig om väldigt stora mängder data. Man anser att man kan acceptera en kvalitetsförsämring för att drastiskt kunna minska storleken hos datafilerna.

En oförstörande kompression tar inte bort någon information utan sparar utrymme genom att representera den på ett alternativt sätt. Detta finner tillämpningar till exempel inom dokumentlagring eller dataöverföring i nätverkskommunikation, då det är mycket viktigt att originalinformationen förblir intakt.

Allmänt utnyttjas i alla typer av kompressionsmetoder att det ofta uppstår mönster i större datamängder. Ett lättförståeligt exempel är en textsträng som innehåller sekvenser av upprepade tecken. Antag att varje tecken representeras av en byte (åtta bitar). Strängen ”aaaabbbbccccdddd” skulle då uppta 16 byte på en hårddisk eller liknande. Detta verkar onödigt mycket, då innehållet i strängen bara är fyra olika tecken. En ZIP-kompression av den typ jag använt i mitt arbete utnyttjar upprepningar i inmatad data och skapar en ny representation som tar upp mindre plats än utrymmet som krävs för rådata.

Enligt principen jag har beskrivit ovan var jag alltså intresserad av att utnyttja eventuella mönster i ljudsignalens amplitudvärden för att minska det utrymme som krävs för att lagra signalen digitalt. För att öka sannolikheten att mönster uppstår försöker jag minska signalens frekvensinnehåll enligt en lågpassprincip – höga frekvenser elimineras för att reducera signalens detaljrikedom. Naturligtvis går det att komprimera en icke-filtrerad signal, men teoretiskt sett bör komprimeringen lyckas bättre om en del av frekvensinnehållet tas bort.

Det inses nu att min kompressionsmetod totalt sett är förstörande – den filtrerar ju bort vissa frekvenser. Dock resonerade jag så att den slutgiltiga dataförlusten ska komma från själva filtreringen och inte från (den kända) kompressionsmetoden. Därför valde jag, som jag nämnde ovan, att använda en kompression av ZIP-typ – den öppna GZIP-kompressionen närmare bestämt. Denna finns färdigimplementerad i Javas klasspaket java.util.zip och är lätt att använda sig av.

För att sammanfatta min kompressionsprincip kan jag ange följande steg:

1. Läs in signaldata från fil
2. Filtrera bort frekvenser högre än en viss brytfrekvens
3. Komprimera signalen med GZIP
4. Spara komprimerad data till en ny fil

2.2 Wave-formatet i korthet

Jag avsåg, som tidigare nämnt, att utveckla ett program som klarade att arbeta med digitala ljudsignaler i formatet Wave. Att läsa in data från en fil i Java är ingen stor sak. För att kunna tolka den måste jag veta hur formatet är upplagt. Detta hade jag ingen kunskap om när jag inledde projektet, men det visade sig inte vara särskilt svårt att hitta relevant information på Internet. Jag valde att läsa ett utdrag från Microsofts specifikation av RIFF Wave-formatet, se referenslistan för webbadress.

En Wave-fil innehåller information om själva formatet såväl som ljudsignalens data. Javas klasspaket javax.sound.sampled innehåller klasser med metoder för att läsa in sådana filer och direkt få tillgång till formatinformation[3] och data individuellt. För mig återstod att tolka själva ljuddata, vilken i sin tur har en representation som varierar från fall till fall. Jag har begränsat mitt program till att hantera monoljud i åtta eller 16 bitar[4].

[3] Här avser jag information såsom byte-ordning, samplingsfrekvens, antal bitar per sampel och så vidare

[4] Med 16-bitars ljud menas att amplitudvärdet för varje sampel representeras av ett 16 bitars tal

För att förklara hur ljudsignalens data kan vara strukturerad i en Wave-fil vill jag visa ett typexempel. Antag att vi har 16-bitars ljud i stereo – då kan en *frame* [5] se ut som figur 1 nedan. I exemplet antas att ljudet är lagrat med ”låg byte-ordning”, det vill säga den minst inflytelserika byten först för varje sampel.

Frame			
Vänster kanal		Höger kanal	
Minst inflytelserik byte	Mest inflytelserik byte	Minst inflytelserik byte	Mest inflytelserik byte

Figur 1. En frame i 16-bitars stereoljud (låg byte-ordning) enligt Wave-formatet

Varje byte innehåller som bekant åtta bitar och därför måste två byte sättas samman för att det egentliga amplitudvärdet ska kunna utläsas. Hopsättningsmetoden beror på vilken byte-ordning det är. I detta exempel skulle ett 16-bitars sampel fås genom att skifta den mest inflytelserika byten åtta bitar åt vänster och sedan sammanfoga den med den minst inflytelserika byten genom en logisk OR-funktion. Java kan naturligtvis hantera sådana bit-operationer så processen blir enkel att genomföra när man känner till principen. Monoljud blir ännu litet enklare än stereo eftersom man inte behöver hålla ordning på de olika kanalerna – det finns per definition bara en.

2.3 Ljudfiltrering i frekvensplanet

Jag nämnde tidigare att min kompression filtrerar originalsignalen enligt en lågpassprincip. Detta valde jag att låta programmet utföra i frekvensplanet. Jag hade även kunnat ta fram ett filters impulssvar och falta insignalen med detta, men eftersom jag vill att användaren ska kunna variera brytfrekvensen från fall till fall skulle detta innebära att programmet måste framställa ett nytt filter för varje körning, vilket inte kändes intressant. Då filtreringen inte behöver ske i realtid tog jag fram en lösning där en frekvensanalys utförs genom diskret Fouriertransform.

Eftersom det är väl invecklat att utveckla en egen FFT och då det inte är syftet med mitt projekt letade jag reda på färdiga metoder på Internet. Jag hittade en endimensionell FFT med motsvarande invers skrivna av Jeffrey D. Taft som jag med hans tillstånd fick använda i projektet. För adress till hans webbsida, se referenslistan.

Min filtreringslösning fungerar så att ljudsignalen delas upp i mindre delar, där varje del innehåller ett visst antal sampel som av tekniska skäl alltid är en tvåpotens[6]. Dessa delar multipliceras med en fönsterfunktion och transformeras i tur och ordning till frekvensplanet, där alla frekvenser (negativa såväl som positiva) över en angiven brytfrekvens elimineras. Nästa steg är att transformera tillbaka och dividera med samma fönsterfunktion som tidigare, samt att slutligen sammanfoga alla delar av signalen till en filtrerad utsignal.

[5] En frame är ett ”totalsampel” vid en viss tid som innehåller amplitudsampel från samtliga ljudkanaler. Den totala signalen består av en lång rad frames som läses från fil efter varandra

[6] FFT-algoritmen bygger på att antalet sampel är jämnt delbart med två rekursivt

Anledningen till att mitt program använder en fönsterfunktion (ett Hanning-fönster för att vara exakt) är att det annars uppstår ”nya” höga frekvenser som en effekt av att varje del av signalen inte är periodisk. Den diskreta Fouriertransformen som utnyttjas vid frekvensanalysen förutsätter att insignalen är periodisk, vilket oftast inte är fallet när det rör sig om kanske några tusen sampel från ett inspelat ljud. Fönsterfunktionen dämpar ned signalen till samma värde i båda ändar så att den upplevs som periodisk och störande nya frekvenser ska därför inte uppstå.

3 Sammanfattning av arbetet

Även om själva programmeringsarbetet inte är det mest intressanta i projektet vill jag nämna litet om det i alla fall. Målet med denna del av rapporten är att läsaren ska få en inblick i hur jag strukturerat mitt program och litet om vilka metoder och klasser som använts. För en total genomgång hänvisar jag till källkoden bifogad i Bilaga A.

3.1 Arbetsgång

När jag hade principen för komprimeringsprocessen klar för mig satte jag mig ned och tänkte igenom vilka funktioner detta medförde att programmet måste ha. De huvudsakliga delmomenten är, i korrekt ordning, datainhämtning (läsa från fil), datakonvertering (anpassning efter vilken datatyp FFT-metoderna vill ha på signalens amplitudvärden), filtrering (inklusive frekvensanalys), datakonvertering (tillbaka till byte-format för lagring) och slutligen datalagring (skriva till fil).

Nästa steg i utvecklingsprocessen var att skriva dessa erfordrade metoder. Jag valde att placera dem alla i en och samma klass, Sound_Data_Handler. Under arbetets gång kunde jag hela tiden testa funktionaliteten genom att modifiera en extern ”styrklass” som fungerade som huvudklass. Denna används inte i slutversionen, då jag istället ”klistrat på” det grafiska gränssnittet som ökar användbarheten dramatiskt.

Som antytt var det grafiska gränssnittet det sista steget i programmeringsarbetet. Gränssnittet är rimligt genomtänkt och jag anser det vara logiskt uppbyggt, även om det lämnar en del att önska. Hur som helst fyller det sitt syfte och det är vad jag hade som krav när jag utvecklade det.

När jag egentligen var klar med programmet kunde jag inte låta bli att lägga till en funktion som tillåter användaren att lyssna på det filtrerade ljudet innan slutgiltig skrivning till fil. Detta är mest en bekvämlighetsfunktion, men det är också roligt att kunna lyssna på det filtrerade resultatet för att kunna höra att resultatet verkligen är det man eftersträvat.

3.2 Programmets klasser – en presentation

Mitt program består huvudsakligen av tre klasser: Sound_Data_Handler, Comp_GUI och FFT. Ytterligare två hjälpklasser – SwingWorker och ExampleFileFilter – används, men dessa tillhandahålls av Sun Microsystems som tilläggsklasser till standardversionen av Java SDK[7] och jag kommer inte att behandla dem här, på samma sätt som jag inte i någon detalj beskriver några av Javas kärnklasser.

3.2.1 Sound_Data_Handler

I klassen Sound_Data_Handler sker det mesta av arbetet. Här finns metoder för läsning och skrivning av filer, filtrering, konvertering mellan byte-datatyp och double-datatyp samt komprimering. Jag kompletterade även klassen med en metod för uppspelning av det filtrerade ljudet. Av de nämnda metoderna är filtreringsprocessen den mest omfattande och använder sig i sin tur av klassen FFT. Sound_Data_Handler håller reda på all data under hela programförloppet, och är därför den överlägset mest minneskrävande komponenten.

3.2.2 Comp_GUI

Comp_GUI är huvudklassen i programmet och innehåller det grafiska gränssnittet. När klassen instansieras skapas ett objekt av typen Sound_Data_Handler som sedan har hand om alla underliggande funktioner enligt tidigare beskrivning. Kortfattat kan man säga att det grafiska gränssnittet är en panel som användaren utnyttjar för att styra programmets funktioner, utan att behöva bekymra sig med hur metoderna fungerar i sig. Förutom att innehålla alla grafiska komponenter tar Comp_GUI även hand om eventuella undantagsfel som kan uppstå under körning av programmet. Jag har valt att låta gränssnittet visa de viktigaste feltyperna för användaren, medan andra döljs. Detta är egentligen att bryta mot god programmeringssed, men syftet med programmet är inte att det grafiska gränssnittet ska vara kommersiellt gångbart.

3.2.3 FFT

Denna klass har jag egentligen inte skrivit själv – dess två metoder är utvecklade av Jeffrey D. Taft och jag använder dem med hans tillstånd. Dock är klassen så viktig att jag ändå vill beskriva dess innehåll, och jag har också valt att ta med den i källkodsbilagan i slutet av rapporten.

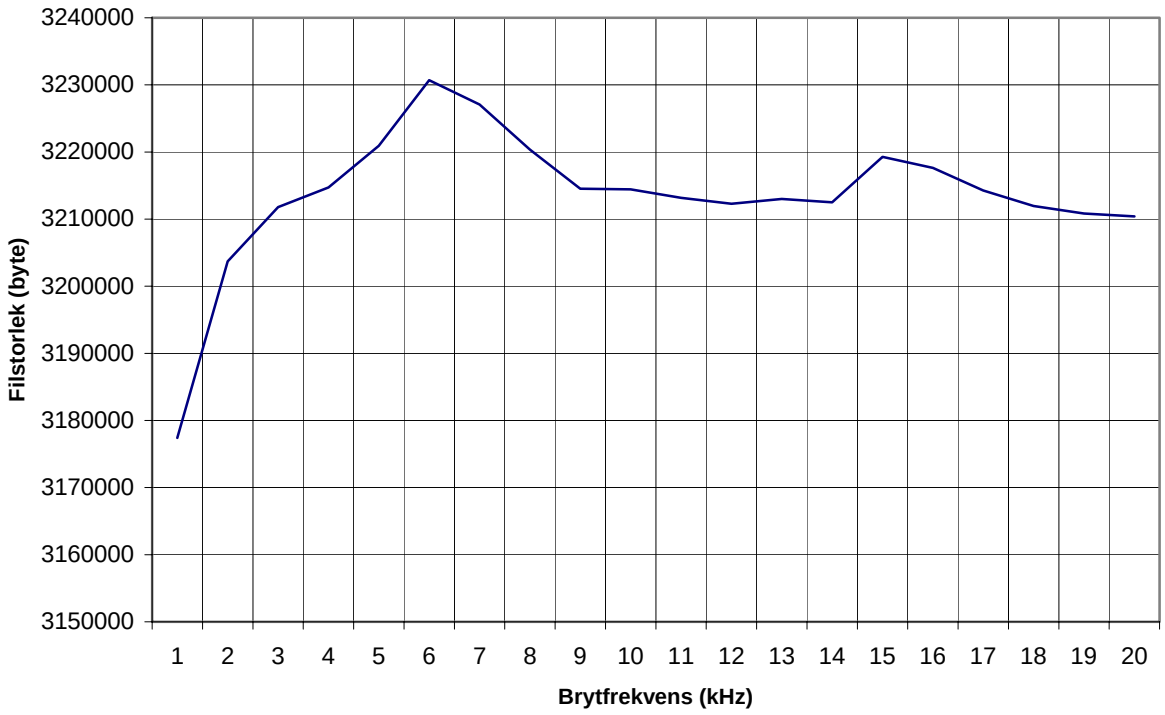
Klassen innehåller metoder för (endimensionell) diskret Fouriertransform implementerad med en FFT-algoritm samt motsvarande invers transform. För mer information, se referenslistan.

[7] Java SDK står för *Java Software Development Kit* och är den programvara/klassuppsättning som behövs för att kunna kompilera Java-källkodsfiler

4 Experiment

Naturligtvis är det intressant att undersöka hur min kompressionsprincip fungerar i praktiken. Jag gjorde ett litet experiment för att verifiera att teorin verkligen stämmer och här redovisar jag resultatet.

Som insignal valde jag en Wave-fil med en samplingsfrekvens på 44100 Hz och med ungefär 40 sekunder långt ljud. Filen upptog 3 545 908 byte och innehöll ganska många frekvenskomponenter. Jag körde filen genom mitt program med några olika brytfrekvenser för filtreringen. I figur 2 nedan redovisas resultatet av mitt experiment.



Figur 2. Filstorlekar efter filtrering/komprimering med varierande brytfrekvens

Som synes verkar kompressionsprincipen fungera bra för brytfrekvenser från ungefär 6 kHz och nedåt. Vid högre brytfrekvenser varierar filstorleken på ett sätt som kan tyckas underligt. Jag tror att detta beror på att de mest betydelsefulla frekvenskomponenterna är relativt låga, och att resultatet därför är beroende av huruvida dessa finns kvar eller ej.

När resultaten tolkas kan det vara bra att tänka på att GZIP-kompressionen som programmet använder egentligen fungerar bäst på textsträngar och liknande. Detta yttrar sig troligen i den låga kompressionsgraden.

5 Insikter

När jag arbetade med mitt program lärde jag mig en hel del, och jag kom fram till flera tydliga nackdelar med min strategi för att lösa filtrerings- och komprimeringsproblemen. Här vill jag presentera några av de svagheter mina lösningar har, och förslag på hur man skulle kunna förbättra implementationen i ett framtida projekt.

5.1 Minnesekonomi

Det ska villigt erkännas att mitt program har en fullständigt vidrig minnesekonomi. Detta innebär i praktiken att det bara är tillämpbart på mindre ljudfiler, vilket jag tycker är acceptabelt eftersom mitt projekt gick ut på att testa principen – inte att utveckla ett program som är gångbart i vardagsbruk.

Grunden till den stora minnesanvändningen ligger i att de FFT-metoder jag använder tar en insignal med sampelvärden av typen Double som argument. Eftersom Double är 64-bitars flyttal innebär detta ett minnesslöseri med en faktor fyra då mitt program kan hantera 16-bitars ljud som högst. Programmet konverterar helt enkelt till exempel 16-bitars amplitudvärden till 64 bitar. Dessutom läser det in hela ljudsignalen i en omgång och lagrar såväl 16-bitars- som 64-bitars-versionen i arbetsminnet. Även en filtrerad upplaga av ljudet lagras. Det är uppenbart att detta kan lösas mer ekonomiskt, exempelvis genom att man läser små avsnitt av ljudet från originalfilen istället för att läsa in hela på en gång.

Minnesproblemet är visserligen påtagligt när man använder programmet, men jag anser att det inte påverkar projektets syfte. Dessutom är det en ren programmeringsteknisk aspekt som jag är fullt medveten om, och jag lämnar det därför därhän.

5.2 Artefakter i utsignalen

Det har visat sig att ljudsignalerna som resulterar av behandling med mitt komprimeringsprogram ibland har irriterande ”klickljud” med jämna tidsintervall. Jag är osäker på hur dessa uppstått, men jag har genom experiment fastslagit att det kan uppstå ett klickljud i varje ”skarv” mellan filtrerade delar av insignalen. Eftersom det är ganska korta delar, 8192 sampel för att vara exakt, skulle det kunna uppstå störfrekvenser på grund av icke-periodicitet i ljudet när programmet utför filtreringen, men detta borde ha eliminerats genom fönsterhanteringen, beskriven tidigare.

Rent teoretiskt kan jag inte förstå var missljuden kommer ifrån. Skarvarna mellan de filtrerade delarna av ljudet är ju egentligen inga skarvar – övergången bör vara helt sömlös. Då inte heller handledaren för projektet har kunnat lista ut var klickljuden kommer ifrån konstaterar jag helt enkelt att de finns där och lämnar öppet för läsaren att fundera över var de kan komma ifrån. Vad jag kan se så har jag inte missuppfattat teorin på något sätt som skulle ligga till grund för effekten.

6 Avslutande ord

I detta projekt antog jag mig en utmanande uppgift som gick ut på att i tillämpa teori från kursen i praktiken genom en implementation av ett lågpassfilter. För att göra något mer än att bara filtrera utnyttjade jag filtreringens effekter för att mer effektivt kunna komprimera ljudsignalen.

Jag är väldigt nöjd när jag ser resultatet av mitt arbete. Framförallt känns det tillfredsställande att den princip för kompression (och ökande av kompressionsgrad genom filtrering) som jag i projektets början resonerade mig fram till visade sig fungera i verkligheten. Det är utan större förvåning men ändå med ett visst mått av fascination jag konstaterar detta.

Mitt program är inte perfekt – det finns nackdelar vad gäller minneshantering, gränssnitt och ljudkvalitet – men jag tycker att det fyller sitt syfte, nämligen att testa mitt resonemang i praktiken. Om jag (eller någon annan) skulle vilja vidareutveckla programmet i framtiden bör det inte vara alltför omständligt att uppdatera de delar av källkoden som krävs för att uppnå önskat resultat.

Slutligen vill jag säga att jag är nöjd med mitt arbete, samt att jag har lärt mig mycket under projektets gång. Dels har jag tillgodogjort mig ökade programmeringstekniska kunskaper och dels har jag fått en djupare förståelse för den delen av kursen som ligger inom ramarna för projektet. I och med detta ser jag kursen som avslutad samtidigt som jag konstaterar att det finns mycket intressant att utforska i det aktuella ämnet!

7 Referenslista

Utdrag ur Microsofts specifikation för ljudformatet RIFF WAVE:
http://www.harmony-central.com/Computer/Programming/wave-format.txt (2004-03-16)

Edward W. Kamen & Bonnie S. Heck
Fundamentals of Signals and Systems using the Web and Matlab
Prentice Hall 2000, andra utgåvan

FFT för Java (färdiga metoder) av Jeffrey D. Taft
http://www.nauticom.net/www/jdtaft (2004-03-16)

SwingWorker (Java-klass), Sun Microsystems
http://java.sun.com/docs/books/tutorial/uiswing/misc/example-1dot4/SwingWorker.java (2004-03-19)

ExampleFileFilter (Java-klass), Sun Microsystems
[Java-katalogen]/demo/jfc/FileChooserDemo/ExampleFileFilter.java
(Filen följer med standarddistributionen av Java SDK)

Bilaga A: Källkod

På sidorna som följer redovisar jag källkoden för mitt program. Förutom dessa tre klasser använder programmet även de två klasserna SwingWorker och ExampleFileFilter, utvecklade och publicerade av Sun Microsystems. För källkoden till dessa, se referenslistan i rapporten.

Jag har valt att låta Appendix A ha en egen sidnumrering, och detta är dess innehåll:

PUBLIC CLASS SOUND_DATA_HANDLER 1
PUBLIC CLASS COMP_GUI 10
PUBLIC CLASS FFT 18

public class Sound_Data_Handler

```
import javax.sound.sampled.*;
import java.io.*;
import javax.swing.*;
import java.util.zip.*;

/**
*  Denna klass tillhandahåller grundläggande funktioner för att läsa in och temporärt lagra ljuddata från en ljudfil.
*  Än så länge stöds endast ljud i 8 eller 16 bitar.
*/

public class Sound_Data_Handler {

   //Variabler
   private int samples_per_step;                    //Int för lagring av hur många samples som ska processas i taget
   private byte[] audio_data;                       //Array för lagring av ljuddata när den lästs från fil
   private byte[] audio_data_filtered;              //Array för lagring av filtrerad ljuddata
   private double[] audio_data_doubles;             //Array för lagring av ljuddata som doubles
   private double[] audio_data_doubles_filtered;    //Array för lagring av filtrerad ljuddata som doubles
   private byte[] audio_data_compressed;            //Array för lagring av komprimerad ljuddata
   private AudioInputStream audio_in_stream;        //Dataström för läsning av ljuddata
   private AudioFormat audio_format;                //Det inlästa ljudets format
   private Deflater deflater;                       //Komprimeringsobjekt
   private DeflaterOutputStream def_stream;         //Dataström för komprimerad data
   private int filtering_status;                    //Indikerar hur långt (0-100) filtreringsprocessen kommit
   private PipedInputStream pipe_in;                //Används för strömning av data vid komprimering/filskrivning
   private PipedOutputStream pipe_out;              //Används för strömning av data vid komprimering/filskrivning
   private int compressed_data_length;              //Anger hur många bytes lång den komprimerade ljuddatan är
   private AudioInputStream audio_filtered_stream;  //Ljudström för filtrerad data
   private SourceDataLine data_line;                //Datakoppling för uppspelning av ljud
   private DataLine.Info data_line_info;            //Infoobjekt för datakopplingen
   private byte[] player_buffer;                    //Buffert för uppspelning av ljud

   /**
      Konstruktor, sätter filtering_status till 0 och skapar ett objekt av typen Sound_Data_Handler
   */
   public Sound_Data_Handler() {
      filtering_status = 0;
   }
```

```
/**
   Läser in ljuddata från angiven ljudfil
   @param in_file Ljudfil att läsa från
   @return Antal bytes som lästes från filen
   @exception FileNotFoundException om filen inte kunde hittas
   @exception IOException om det inte gick att läsa från filen
   @exception UnsupportedAudioFileException om filen inte har ett giltigt format
*/
public int read_data(File in_file) throws FileNotFoundException, IOException, UnsupportedAudioFileException {
   //Lagra information om ljudfilens format
   audio_format = AudioSystem.getAudioFileFormat(in_file).getFormat();

   //Kontrollera om filen har ogiltigt format
   if ((audio_format.getSampleSizeInBits() != 8 && audio_format.getSampleSizeInBits() != 16) || audio_format.getChannels() != 1) {
      audio_format = null;
      throw new UnsupportedAudioFileException("Unsupported sample size");
   } else {

      //Koppla filen till dataströmmen
      audio_in_stream = AudioSystem.getAudioInputStream(in_file);

      //Ställ in arrayens storlek
      audio_data = new byte[(int)(audio_in_stream.getFrameLength() * audio_format.getFrameSize())];

      //Läs in hela ljudfilen via ljudströmmen och lagra data i arrayen
      return audio_in_stream.read(audio_data);
   }
}

/**
   Skriver komprimerad ljuddata till angiven fil
   @param out_file Ljudfil att skriva till
   @exception FileNotFoundException om filen inte kunde hittas
   @exception IOException om det inte gick att skriva till filen
   @exception Exception om det inte finns tillgänglig data eller format
*/
public void write_data(File out_file) throws FileNotFoundException, IOException, Exception {
   if (audio_format == null || audio_data == null || audio_data_compressed == null)
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() first");
   else {
      //Skapa en ström för den komprimerade datan
      ByteArrayInputStream comp_in = new ByteArrayInputStream(audio_data_compressed, 0, compressed_data_length);

      //Skapa en inputström kopplad till komprimerad ljuddata med rätt format
      AudioInputStream out_stream = new AudioInputStream(comp_in, audio_format, compressed_data_length / audio_format.getFrameSize());

      //Skriv till fil
      AudioSystem.write(out_stream, AudioFileFormat.Type.WAVE, out_file);
   }
}
```

```
/**
   Spelar upp den aktuella filtrerade signalen, om det finns någon
   @exception IOException om det inte gick att skriva till filen
   @exception Exception om det inte finns tillgänglig data eller format
*/
public void play_filtered_audio() throws IOException, Exception {
   if (audio_format == null || audio_data_filtered == null)
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() first");
   else {
      try {
         //Skapa en ström för den filtrerade datan
         ByteArrayInputStream temp = new ByteArrayInputStream(audio_data_filtered);

         //Skapa en inputström kopplad till filtrerad ljuddata med rätt format
         audio_filtered_stream = new AudioInputStream(temp, audio_format, audio_data_filtered.length / audio_format.getFrameSize());

         //Skapa infoobjekt för datakopplingen
         data_line_info = new DataLine.Info(SourceDataLine.class, audio_format);

         //Skapa datakopplingen
         data_line = (SourceDataLine) AudioSystem.getLine(data_line_info);

         //Öppna kopplingen så att den kan ta emot data för uppspelning
         data_line.open(audio_format);

         //Aktivera kopplingen så att den skickar vidare data till högtalarna
         data_line.start();

         //Spela upp ljudet genom att stega igenom den filtrerade datan och skicka den till kopplingen
         int   no_bytes_read = 0;              //Indikerar hur många byte som lästes senast
         player_buffer = new byte[128000];     //Skapa uppspelningsbuffert
         while (no_bytes_read != -1)
         {
            //Läs från filtrerad data
            no_bytes_read = audio_filtered_stream.read(player_buffer, 0, player_buffer.length);

            if (no_bytes_read >= 0)
            {
               //Skriv till datakopplingen (och vidare ut till högtalarna)
               int   no_bytes_written = data_line.write(player_buffer, 0, no_bytes_read);
            }
         }
```

```
         //Låt ljudet spela klart
         data_line.drain();

         //Stäng datakopplingen
         data_line.close();

      } catch (LineUnavailableException line_ex) {
      }
   }
}

/**
   Komprimerar filtrerad ljuddata och lagrar denna
   @exception Exception om det inte finns tillgänglig data eller format
*/
public void compress_data() throws Exception {
   if (audio_format == null || audio_data == null || audio_data_filtered == null)
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() first");
   else {
      //Instansiera en Deflater inställd på bästa (långsammaste) kompression
      deflater = new Deflater(Deflater.BEST_COMPRESSION, true);

      //Reservera minne för den komprimerade datan
      audio_data_compressed = new byte[audio_data_filtered.length];

      //Ställ in den filterade datan som input för deflatern
      deflater.setInput(audio_data_filtered);
      deflater.finish();

      //Komprimera data och "kom ihåg" hur många bytes den tar upp
      int data_length = deflater.deflate(audio_data_compressed);

      //Det är inte säkert att antal bytes i den komprimerade byte-arrayen
      //är en jämn multipel av frame-storleken. Detta bör det vara för att
      //skrivningen till fil ska kunna ske utan risk för fel. Här ordnas
      //detta:
      compressed_data_length = (int)Math.ceil((double)data_length / (double)audio_format.getFrameSize()) * audio_format.getFrameSize();
   }
}

/**
   Sätter hur långt filtreringsprocessen har kommit
   @param status Position i processen (0-100)
*/
public void set_filtering_status(int status) {
   filtering_status = (status >= 0 && status <= 100) ? status : 0;
}
```

```
/**
   Returnerar hur långt filtreringsprocessen har kommit
   @return Position i processen (0-100)
*/
public int get_filtering_status() {
   return filtering_status;
}

/**
   Returnerar den data som finns lagrad, om någon
   @return Ljuddata som array av bytes
   @exception Exception om ingen data tidigare lästs in
*/
public byte[] get_audio_data() throws Exception {
   if (audio_data == null)
      throw new Exception("No data avalable. Please use Sound_Data_Handler.read_data() first");
   else
      return audio_data;
}

/**
   Returnerar ljudformatet för den data som finns lagrad, om någon
   @return Formatet på ljuddata
   @exception Exception om det inte finns tillgängligt format
*/
public AudioFormat get_audio_format() throws Exception {
   if (audio_format == null)
      throw new Exception("No format avalable. Please use Sound_Data_Handler.read_data() first");
   else
      return audio_format;
}

/**
   Returnerar ljuddataströmmen för den data som finns lagrad, om någon
   @return Ljuddataströmmen
   @exception Exception om ingen data tidigare lästs in
*/
public AudioInputStream get_audio_input_stream() throws Exception {
   if (audio_in_stream == null)
      throw new Exception("No stream avalable. Please use Sound_Data_Handler.read_data() first");
   else
      return audio_in_stream;
}
```

```
/**
   Konverterar inläst ljuddata till doubles och lagrar dem
   @return Arrayen av doubles
   @exception Exception om det inte finns tillgänglig data eller format
*/
public double[] bytes_to_doubles() throws Exception {
   if (audio_format == null || audio_data == null)
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() first");
   else {

      audio_data_doubles = new double[audio_data.length/(audio_format.getSampleSizeInBits()/8)];

      //Gå igenom byte-arrayen och konvertera och lagra data i double-arrayen
      if (audio_format.getSampleSizeInBits() == 8) {

         //Om ljudfilens format representerar data med 8 bitar är det lätt - konvertera bara till double:
         for (int i = 0; i < audio_data.length; i++) {
            audio_data_doubles[i] = (double)(audio_data[i] & 0xFF);
         }

      } else {

         //Om ljudfilens format representerar data med 16 bitar är litet mer jobb:
         for (int i = 0; i < (audio_data.length/2); i++) {

            //Tolka data olika beroende på byte-order
            if (audio_format.isBigEndian()) {
               audio_data_doubles[i] = (double)(short)(((audio_data[2*i] & 0xFF) << 8) | (audio_data[2*i+1] & 0xFF));
            } else {
               audio_data_doubles[i] = (double)(short)(((audio_data[2*i+1] & 0xFF) << 8) | (audio_data[2*i] & 0xFF));
            }
         }
      }
   }
   return audio_data_doubles;
}
```

```
/**
   Konverterar double-ljuddata till bytes och lagrar
   @return Arrayen av bytes
   @exception Exception om ingen data tidigare lästs in och filtrerats
*/
public byte[] doubles_to_bytes() throws Exception {
   if (audio_format == null || audio_data == null || audio_data_doubles_filtered == null)
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() first");
   else {

      audio_data_filtered = new byte[audio_data.length];

      //Gå igenom double-arrayen och konvertera och lagra data i byte-arrayen
      if (audio_format.getSampleSizeInBits() == 8) {
         //Om ljudfilens format representerar data med 8 bitar är det lätt - konvertera bara till byte:
         for (int i = 0; i < audio_data.length; i++) {
            audio_data_filtered[i] = (byte)audio_data_doubles_filtered[i];
         }

      } else {

         //Om ljudfilens format representerar data med 16 bitar är litet mer jobb:
         for (int i = 0; i < (audio_data_doubles_filtered.length); i++) {

            //Tolka data olika beroende på byte-order
            if (audio_format.isBigEndian()) {
               audio_data_filtered[2*i] = (byte)(((short)audio_data_doubles_filtered[i] & 0xFF00) >> 8);
               audio_data_filtered[2*i+1] = (byte)((short)audio_data_doubles_filtered[i] & 0x00FF);
            } else {
               audio_data_filtered[2*i] = (byte)((short)audio_data_doubles_filtered[i] & 0x00FF);
               audio_data_filtered[2*i+1] = (byte)(((short)audio_data_doubles_filtered[i] & 0xFF00) >> 8);
            }
         }
      }
   }
   return audio_data_filtered;
}
```

```
/**
   Går igenom den inlästa ljuddatan och gör frekvensanalys samt filtrerar bort frekvenser högre än en viss brytfrekvens
   @param sa_step Antal sampel att analysera åt gången, måste vara sa_step = 2^n där n = positivt heltal
   @param cut_freq Brytfrekvens i Hz
   @return En komplex array med den filtrerade signalen
   @exception Exception om ingen data tidigare lästs in och konverterats till doubles
*/
public double[] filter_audio_data(int sa_step, double cut_freq) throws Exception {
   if (audio_data == null || audio_format == null || audio_data_doubles == null) {
      throw new Exception("No format/no data available. Please use Sound_Data_Handler.read_data() and -bytes_to_doubles() first");
   } else {
      //Se till så att sa_step verkligen är 2^n, n positivt heltal
      samples_per_step = (int)Math.ceil(Math.pow(2d, (double)(Math.log(sa_step)/Math.log(2))));

      if (cut_freq >= audio_format.getSampleRate()/2) {
         //Om brytfrekvensen är högre än eller lika med halva samplingsfrekvensen - utför ingen filtrering
         audio_data_doubles_filtered = audio_data_doubles;
      } else {
         //Gör i ordning en array för den filtrerade ljudsignalen
         audio_data_doubles_filtered = new double[audio_data_doubles.length];

         //Räkna ut i hur många steg filtreringen kommer att ske
         int steps = (int)Math.ceil(audio_data_doubles.length / samples_per_step);

         //Gör i ordning en temporär array för lagring av varje omgång av sampel
         double[][] samples = new double[samples_per_step][2];

         //Beräkna vilken start- och slutposition brytfrekvensen motsvarar
         int start_cut = (int)Math.round(((double)samples_per_step / 2d) * cut_freq / (audio_format.getSampleRate() / 2d));
         int stop_cut = samples_per_step - start_cut;

         //Skapa ett Hanning-fönster av rätt storlek
         double[] hamming_win = new double[samples_per_step];
         for (int i = 0; i < samples_per_step; i++) {
            hamming_win[i] = 0.54 - 0.46 * Math.cos((2 * Math.PI * i) / (samples_per_step - 1));
         }

         //Filtrera signalen steg för steg
         for (int step = 0; step < steps; step++) {

            //Kopiera aktuella sampel
            for (int pos = 0; pos < samples_per_step; pos++) {
               //Om signalens längd överskrids detta steg - kompensera
               if (pos + step * samples_per_step >= audio_data_doubles.length) {
                  samples[pos][0] = 0.08;

               } else {
                  //Skala med Hamming-fönstret och lägg in det erhållna värdet
                  samples[pos][0] = (audio_data_doubles[pos + step * samples_per_step]) * hamming_win[pos];
               }
```

```
                    samples[pos][1] = 0;
                }

                //Fouriertransformera samplen
                samples = FFT.fft_1d(samples);

                //Sätt alla frekvenskomponenter i stoppbandet till 0
                for (int pos = start_cut; pos < stop_cut; pos++) {
                    samples[pos][0] = 0;
                    samples[pos][1] = 0;
                }

                //Skala de Fouriertransformerade samplen med antalet sampel
                for (int pos = 0; pos < samples_per_step; pos++) {
                    samples[pos][0] /= samples_per_step;
                    samples[pos][1] /= samples_per_step;
                }

                //Inverstransformera samplen
                samples = FFT.ifft_1d(samples);

                //Sätt in de filtrerade samplen (omvänt skalade med Hanningfönstret) i den totala filtrerade signalen
                for (int pos = 0; pos < samples_per_step; pos++) {
                    if (pos + step * samples_per_step < audio_data_doubles_filtered.length) {
                        audio_data_doubles_filtered[pos + step * samples_per_step] = Math.round(samples[pos][0]) / hamming_win[pos];
                    }
                }

                //Uppdatera processstatus
                filtering_status = (int)Math.round((double)step/(double)steps*100d);
            }
        }
        filtering_status = 100;
        return audio_data_doubles_filtered;
    }
  }
}
```

public class Comp_GUI

```
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.sound.sampled.*;

/**
*  Denna klass tillhandahåller ett grafiskt gränssnitt för ljudfiltrering och -komprimering.
*/
public class Comp_GUI extends JFrame implements ActionListener, ChangeListener, WindowListener {
   //Grafiska komponenter
   private JProgressBar progress;                              //Förloppsindikator för filtreringsprocessen
   private JFileChooser file_chooser, save_file_chooser;         //Filväljarfönster för öppna/spara-dialog
   private JSlider cut_freq_slider;                          //Slider för val av brytfrekvens
   private JPanel content_pane;                              //Innehållslagret för fönstret
   private JButton file_open_button, file_save_button, filter_button, comp_save_button, play_button; //Diverse knappar
   private JPanel input_info;                                //Lager för information om insignalen
   private JLabel stat_samples, stat_channels, stat_samplerate, stat_samplebits;   //Rubriklabels för information om insignalen
   private JLabel info_samples, info_channels, info_samplerate, info_samplebits;   //Textlabels för information om insignalen
   private JLabel input_file_label, output_file_label;             //Labels för in- och outputfiltext
   private JLabel slider_text;                                  //Text till brytfrekvens-slider
   private JLabel current_cut_freq;                          //Label för att visa vald brytfrekvens

   //Övriga objekt
   private File input_file, output_file;       //Filer för in- och output
   private Sound_Data_Handler data_handler;    //Själva datahanteringsobjektet som filtererar signalen
   private Timer progress_timer;

   //Teckensnitt
   public static final Font INFO_LABEL_FONT = new Font("Arial", Font.BOLD, 11);
   public static final Font INFO_TEXT_FONT = new Font("Arial", Font.PLAIN, 11);
   public static final Font TINY_TEXT_FONT = new Font("Arial", Font.PLAIN, 8);
```

```
/**
   Initierar ett grafiskt fönster som fungerar som gränssnitt för programmet
*/
public Comp_GUI() {
   //Instansiera de grafiska komponenterna
   progress = new JProgressBar();
   file_chooser = new JFileChooser();
   save_file_chooser = new JFileChooser();
   cut_freq_slider = new JSlider(1, 11025);
   content_pane = new JPanel();
   input_info = new JPanel();
   file_open_button = new JButton("Bläddra...");
   file_save_button = new JButton("Bläddra");
   filter_button = new JButton("Filtrera");
   stat_samples = new JLabel("Antal frames:");
   stat_channels = new JLabel("Antal kanaler:");
   stat_samplerate = new JLabel("Samplingsfrekvens (Hz):");
   stat_samplebits = new JLabel("Bits per sampel:");
   info_samples = new JLabel("-");
   info_channels = new JLabel("-");
   info_samplerate = new JLabel("-");
   info_samplebits = new JLabel("-");
   input_file_label = new JLabel("Fil för insignal:");
   output_file_label = new JLabel("Fil för utsignal:");
   slider_text = new JLabel("Brytfrekvens (Hz):");
   current_cut_freq = new JLabel("-");
   comp_save_button = new JButton("Komprimera och spara");
   play_button = new JButton("Spela upp");

   //Ställ in teckensnitt
   stat_samples.setFont(INFO_LABEL_FONT);
   stat_channels.setFont(INFO_LABEL_FONT);
   stat_samplerate.setFont(INFO_LABEL_FONT);
   stat_samplebits.setFont(INFO_LABEL_FONT);
   info_samples.setFont(INFO_TEXT_FONT);
   info_channels.setFont(INFO_TEXT_FONT);
   info_samplerate.setFont(INFO_TEXT_FONT);
   info_samplebits.setFont(INFO_TEXT_FONT);

   //Ställ in vilka filer som ska kunna visas i filväljardialogen och filsparardialogen
   file_chooser.addChoosableFileFilter(new ExampleFileFilter("wav", "WAVE Sound file"));
   save_file_chooser.addChoosableFileFilter(new ExampleFileFilter("lpc", "Low-pass Sound Compression file"));

   //Ställ in startkatalog för filväljardialogen
   file_chooser.setCurrentDirectory(new File("."));

   //Initiera datahanteraren
   data_handler = new Sound_Data_Handler();

   //Instansiera och ställ in timern kopplad till förloppsindikatorn
```

```
progress_timer = new Timer(100, this);

//Ställ in manuell layouthantering
content_pane.setLayout(null);
input_info.setLayout(null);

//Lägg till grafiska komponenter till input_info-lagret
input_info.add(stat_samples);
input_info.add(stat_channels);
input_info.add(stat_samplerate);
input_info.add(stat_samplebits);
input_info.add(info_samples);
input_info.add(info_channels);
input_info.add(info_samplerate);
input_info.add(info_samplebits);

//Placera de grafiska komponenterna i inforutan och ställ in deras storlek
stat_samples.setBounds(10, 30, 140, 15);
stat_channels.setBounds(10, 45, 140, 15);
stat_samplerate.setBounds(10, 60, 140, 15);
stat_samplebits.setBounds(10, 75, 140, 15);
info_samples.setBounds(160, 30, 120, 15);
info_channels.setBounds(160, 45, 120, 15);
info_samplerate.setBounds(160, 60, 120, 15);
info_samplebits.setBounds(160, 75, 120, 15);

//Ställ in ram för inforutan
input_info.setBorder(BorderFactory.createTitledBorder("Data för insignal"));

//Lägg till grafiska komponenter till content_pane
content_pane.add(input_file_label);
content_pane.add(output_file_label);
content_pane.add(file_open_button);
content_pane.add(file_save_button);
content_pane.add(input_info);
content_pane.add(slider_text);
content_pane.add(current_cut_freq);
content_pane.add(cut_freq_slider);
content_pane.add(filter_button);
content_pane.add(progress);
content_pane.add(play_button);
content_pane.add(comp_save_button);

//Placera de grafiska komponenterna och ställ in deras storlek
input_file_label.setBounds(100, 5, 150, 20);
file_open_button.setBounds(190, 5, 100, 20);
output_file_label.setBounds(100, 35, 150, 20);
file_save_button.setBounds(190, 35, 100, 20);
input_info.setBounds(5, 70, 280, 110);
slider_text.setBounds(10, 175, 110, 30);
```

```
    current_cut_freq.setBounds(120, 175, 150, 30);
    cut_freq_slider.setBounds(5, 205, 280, 50);
    filter_button.setBounds(10, 270, 90, 20);
    progress.setBounds(105, 270, 185, 20);
    play_button.setBounds(10, 300, 90, 20);
    comp_save_button.setBounds(105, 300, 185, 20);

    //Inställningar för förloppsindikatorn
    progress.setStringPainted(true);

    //Inställningar för slidern
    cut_freq_slider.setMajorTickSpacing(1000);
    cut_freq_slider.setMinorTickSpacing(200);
    cut_freq_slider.setPaintLabels(false);
    cut_freq_slider.setPaintTicks(true);

    //Startinställningar för komponenterna
    file_save_button.setEnabled(false);
    filter_button.setEnabled(false);
    cut_freq_slider.setEnabled(false);
    play_button.setEnabled(false);
    comp_save_button.setEnabled(false);

    //Koppla lyssnare
    this.addWindowListener(this);
    file_open_button.addActionListener(this);
    file_save_button.addActionListener(this);
    cut_freq_slider.addChangeListener(this);
    filter_button.addActionListener(this);
    play_button.addActionListener(this);
    comp_save_button.addActionListener(this);

    //Räkna ut x- och y-positon för fönstrets övre vänstra hörn
    int x_position = (int)Toolkit.getDefaultToolkit().getScreenSize().getWidth() / 2 - 150;
    int y_position = (int)Toolkit.getDefaultToolkit().getScreenSize().getHeight() / 2 - 180;

    //Allmänna fönsterinställningar
    this.setContentPane(content_pane);
    this.setLocation(x_position, y_position);
    this.setSize(300, 360);
    this.setResizable(false);
    this.setTitle("Low-pass Sound Compression");
    this.setVisible(true);
}
```

```
/**
   Uppdaterar formatinforutan
   @param format Formatobjekt
   @param stream AudioInputStream
*/
public void update_format_info(AudioFormat format, AudioInputStream stream) {
   info_samples.setText(new Long(stream.getFrameLength()).toString());
   info_channels.setText(new Integer(format.getChannels()).toString());
   info_samplerate.setText(new Float(format.getSampleRate()).toString());
   info_samplebits.setText(new Integer(format.getSampleSizeInBits()).toString());
}

/**
   Uppdaterar sliderns extremlägen
   @param format Formatobjekt
*/
public void update_slider_minmax(AudioFormat format) {
   cut_freq_slider.setMinimum(1);
   cut_freq_slider.setMaximum((int)Math.floor(format.getSampleRate()/2)-1);
}

/**
   Hanterar knapptryckningar och liknande
   @param e ActionEvent som beskriver händelsen
*/
public void actionPerformed(ActionEvent e) {
   //Kolla vilken komponent som givit upphov till händelsen och agera därefter
   if (e.getSource() == file_open_button) {
      //Välj inputfil
      int choice_return = file_chooser.showOpenDialog(this);
      if (choice_return == JFileChooser.APPROVE_OPTION) {
         //Användaren tryckte på OK - gå vidare
         try {
            String temp_filename = file_chooser.getSelectedFile().getName();
            String temp_filetype = temp_filename.substring(temp_filename.length() - 4, temp_filename.length());
            if (temp_filetype.equals(".wav")) {
               input_file = file_chooser.getSelectedFile();
               data_handler.read_data(input_file);
               update_format_info(data_handler.get_audio_format(), data_handler.get_audio_input_stream());
               file_save_button.setEnabled(true);
               save_file_chooser.setCurrentDirectory(file_chooser.getCurrentDirectory());
            }
         } catch (UnsupportedAudioFileException file_ex) {
            JOptionPane.showMessageDialog(this, "Fel filtyp - endast 8/16-bitars mono WAVE stöds.");
         } catch (Exception ex) {
         }
      }
   }

   if (e.getSource() == file_save_button) {
```

```
   try {
      //Välj outputfil
      int choice_return = save_file_chooser.showSaveDialog(this);
      if (choice_return == JFileChooser.APPROVE_OPTION) {
         //Användaren tryckte på OK - gå vidare
         output_file = new File(save_file_chooser.getSelectedFile() + ".lpc");
         update_slider_minmax(data_handler.get_audio_format());
         cut_freq_slider.setEnabled(true);
         filter_button.setEnabled(true);
      }
   } catch (OutOfMemoryError mem_ex) {
      JOptionPane.showMessageDialog(this, "Slut på arbetsminne!");
      System.exit(0);
   } catch (Exception ex) {
   }
}

if (e.getSource() == filter_button) {
   try {
      data_handler.set_filtering_status(0);
      progress_timer.start();
      filter_button.setEnabled(false);
      cut_freq_slider.setEnabled(false);
      file_open_button.setEnabled(false);
      file_save_button.setEnabled(false);

      //Skapa en SwingWorker för filtreringsprocessen eftersom den kan ta tid
      final SwingWorker worker = new SwingWorker() {
         public Object construct() {
            try {
               data_handler.bytes_to_doubles();
               data_handler.filter_audio_data(8192, cut_freq_slider.getValue());

            } catch (OutOfMemoryError mem_ex) {
               JOptionPane.showMessageDialog(new JFrame(), "Slut på arbetsminne!");
               System.exit(0);
            } catch (Exception exeption) {
            }

            return new String(); //Returnera något som inte används
         }
      };
      worker.start();
   } catch (Exception ex) {
   }
}

if (e.getSource() == progress_timer) {
   //Uppdatera förloppsindikatorn
   progress.setValue(data_handler.get_filtering_status());
```

```
            if (progress.getValue() == 100) {
               progress_timer.stop();
               cut_freq_slider.setEnabled(true);
               file_open_button.setEnabled(true);
               file_save_button.setEnabled(true);
               filter_button.setEnabled(true);
               play_button.setEnabled(true);
               comp_save_button.setEnabled(true);
            }
         }

         if (e.getSource() == play_button) {
            //Spela upp det komprimerade ljudet
            try {
               data_handler.doubles_to_bytes();
               data_handler.play_filtered_audio();
            } catch (OutOfMemoryError mem_ex) {
               JOptionPane.showMessageDialog(this, "Slut på arbetsminne!");
               System.exit(0);
            } catch (Exception ex) {
            }
         }

         if (e.getSource() == comp_save_button) {
            //Komprimera och spara
            try {
               data_handler.doubles_to_bytes();
               data_handler.compress_data();
               data_handler.write_data(output_file);
               JOptionPane.showMessageDialog(this, "Filen sparades!");
            } catch (OutOfMemoryError mem_ex) {
               JOptionPane.showMessageDialog(this, "Slut på arbetsminne!");
               System.exit(0);
            } catch (Exception ex) {
            }
         }

      }
```

```
   /**
      Uppdaterar visualiseringen av aktuell brytfrekvens beroende på sliderns läge
      @param e ChangeEvent som beskriver händelsen
   */
   public void stateChanged(ChangeEvent e) {
      //Kolla vilken komponent som genererat händelser och agera därefter
      if (e.getSource() == cut_freq_slider) {
         current_cut_freq.setText(new Integer(cut_freq_slider.getValue()).toString());
      }
   }

   /**
      Stänger det grafiska fönstret och därigenom hela programmet
      @param e WindowEvent som beskriver händelsen
   */
   public void windowClosing(WindowEvent e) {
      this.setVisible(false);
      this.dispose();
      System.exit(0);
   }

   /*------------------------------------------------------------
   Ej implementerade metoder från interfacet WindowListener
   ------------------------------------------------------------*/
   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}

   /*------------------------------------------------------------

   /**
      Startar programmet och visar det grafiska gränssnittet
   */
   public static void main(String[] args) {
      Comp_GUI gui = new Comp_GUI();
   }
}
```

public class FFT

```
import java.awt.*;

/*
   Metoderna i den här klassen är skrivna av Jeffrey D. Taft. För mer information, se http://www.nauticom.net/www/jdtaft.
   The methods in this class are written by Jeffrey D. Taft. For more information, see http://www.nauticom.net/www/jdtaft.
*/

/*
   This is the Java source code for an FFT routine.
   The array length must be a power of two.
   The array size is [L][2], where each sample is complex;
   array[n][0] is the real part, array[n][1] is the imaginary part of sample n.
*/

public class FFT {

   public static double[][] fft_1d(  double[][] array )
   {
      double  u_r,u_i, w_r,w_i, t_r,t_i;
      int     ln, nv2, k, l, le, le1, j, ip, i, n;

      n = array.length;
         ln = (int)( Math.log( (double)n )/Math.log(2) + 0.5 );
         nv2 = n / 2;
         j = 1;
      for (i = 1; i < n; i++ )
         {
         if (i < j)
         {
               t_r = array[i - 1][0];
               t_i = array[i - 1][1];
               array[i - 1][0] = array[j - 1][0];
               array[i - 1][1] = array[j - 1][1];
               array[j - 1][0] = t_r;
               array[j - 1][1] = t_i;
         }
         k = nv2;
         while (k < j)
         {
               j = j - k;
               k = k / 2;
         }
         j = j + k;
         }
```

```
   for (l = 1; l <= ln; l++) /* loops thru stages */
      {
         le = (int)(Math.exp( (double)l * Math.log(2) ) + 0.5 );
      le1 = le / 2;
      u_r = 1.0;
      u_i = 0.0;
      w_r =  Math.cos( Math.PI / (double)le1 );
      w_i = -Math.sin( Math.PI / (double)le1 );
      for (j = 1; j <= le1; j++) /* loops thru 1/2 twiddle values per stage */
      {
            for (i = j; i <= n; i += le) /* loops thru points per 1/2 twiddle */
            {
            ip = i + le1;
            t_r = array[ip - 1][0] * u_r - u_i * array[ip - 1][1];
            t_i = array[ip - 1][1] * u_r + u_i * array[ip - 1][0];

            array[ip - 1][0] = array[i - 1][0] - t_r;
            array[ip - 1][1] = array[i - 1][1] - t_i;

            array[i - 1][0] =  array[i - 1][0] + t_r;
            array[i - 1][1] =  array[i - 1][1] + t_i;
            }
            t_r = u_r * w_r - w_i * u_i;
            u_i = w_r * u_i + w_i * u_r;
            u_r = t_r;
      }
      }
   return array;
}



/*
   This is the Java source code for an inverse FFT routine.
   The array length must be a power of two.
   The array size is [L][2], where each sample is complex;
   array[n][0] is the real part, array[n][1] is the imaginary part of sample n.

*/

public static double[][] ifft_1d(  double[][] array )
{
   double  u_r,u_i, w_r,w_i, t_r,t_i;
   int     ln, nv2, k, l, le, le1, j, ip, i, n;

   n = array.length;
      ln = (int)( Math.log( (double)n )/Math.log(2) + 0.5 );
      nv2 = n / 2;
      j = 1;
```

```
      for (i = 1; i < n; i++ )
         {
         if (i < j)
         {
               t_r = array[i - 1][0];
               t_i = array[i - 1][1];
               array[i - 1][0] = array[j - 1][0];
               array[i - 1][1] = array[j - 1][1];
               array[j - 1][0] = t_r;
               array[j - 1][1] = t_i;
         }
         k = nv2;
         while (k < j)
         {
               j = j - k;
               k = k / 2;
         }
         j = j + k;
         }

      for (l = 1; l <= ln; l++) /* loops thru stages */
         {
            le = (int)(Math.exp( (double)l * Math.log(2) ) + 0.5 );
         le1 = le / 2;
         u_r = 1.0;
         u_i = 0.0;
         w_r =  Math.cos( Math.PI / (double)le1 );
         w_i =  Math.sin( Math.PI / (double)le1 );
         for (j = 1; j <= le1; j++) /* loops thru 1/2 twiddle values per stage */
         {
               for (i = j; i <= n; i += le) /* loops thru points per 1/2 twiddle */
               {
               ip = i + le1;
               t_r = array[ip - 1][0] * u_r - u_i * array[ip - 1][1];
               t_i = array[ip - 1][1] * u_r + u_i * array[ip - 1][0];

               array[ip - 1][0] = array[i - 1][0] - t_r;
               array[ip - 1][1] = array[i - 1][1] - t_i;

               array[i - 1][0] =  array[i - 1][0] + t_r;
               array[i - 1][1] =  array[i - 1][1] + t_i;
               }
               t_r = u_r * w_r - w_i * u_i;
               u_i = w_r * u_i + w_i * u_r;
               u_r = t_r;
         }
         }
      return array;
   }
}
```