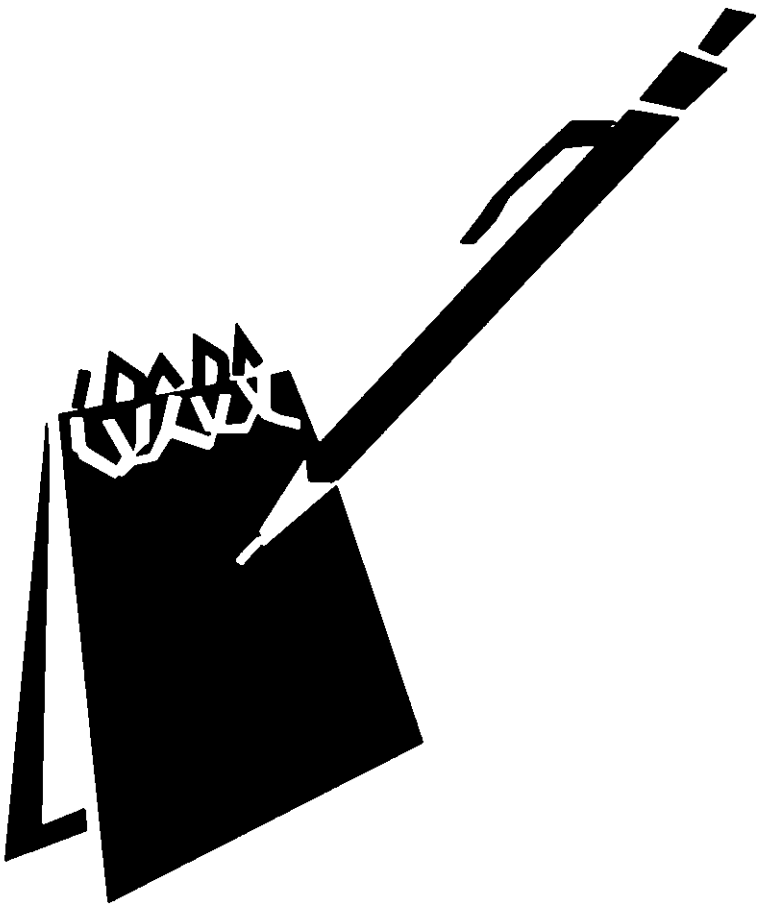
27

Introduction to Audio Compression and Representation



25th International Conference on Computer Graphics and Interactive Techniques
Exhibition **21 23 July** 1998 Conference **19 24 July** 1998
Orlando, Florida USA

course notes

27

Introduction to Audio Compression and Representation

Organizer and Lecturer
Perry Cook
Princeton University

25th International Conference on Computer Graphics and Interactive Techniques
Exhibition **21 23 July** 1998 Conference **19 24 July** 1998
Orlando, Florida USA

course notes

SIGGRAPH 1998 Course #27

Introduction to Audio Compression and Representation

Monday 10 am noon

Description

This course begins with an introduction to waveform sampling and transmission and storage issues followed by an introduction to psycho acoustics that covers the unique acuities and limits of the human auditory system Waveform compression model based speech compression and psychoacoustically based frequency domain compression algorithms are covered MIDI and other music representation schemes are introduced in the context of existing music storage and manipulation systems as well as potential future compression schemes Emphasis is on the tradeoffs of quality compressed data size and flexible data manipulation C code examples are made available for various compression algorithms

Prerequisites

Familiarity with the concepts of sampling and aliasing Specific knowledge of Fourier or wavelet transforms is not required but familiarity with the time and frequency domain representations of data is helpful

Topics Covered

Statistically based compression of audio waveform data is basically useless and loss less compression of audio is generally impossible above the very lowest compression ratios Incorporation of psycho acoustic principles in a perceptually loss less system makes higher compression ratios possible Sound examples are played to demonstrate the tradeoffs of different algorithms and compression ratios

Organizer and Lecturer

Perry Cook
Princeton University

Schedule

10 00 Short overview of compression in general

10 05 Waveform sampling storage and transmission

10 15 Psychoacoustics some limits of auditory perception

10 35 Sound and Music Representation MIDI General MIDI more

11 55 Survey of audio compression algorithms
- a) a law/u law
- b) ADPCM
- c) Perceptually based transform coders (such as MPEG 2 AC2/3 etc)
- d) Production model based coders
- e) The future parametric multi model compression?

11 25 Some details of two specific audio compression algorithms
- a) A production model based speech coder
- b) A perceptually based transform coder

11 45 Wrap up
- a) bibliography of references
- b) source code on CDROM

Presenter Bio

Perry R Cook
Assistant Professor
Department of Computer Science
also Department of Music
Princeton University
35 Olden Street Princeton NJ 08544
609 258 4951 fax 609 258 1771
prc@cs princeton edu

Perry Cook received a BA in music from the University of Missouri at Kansas City Conservatory of Music a BSEE from the University of Missouri Engineering School and Masters and PhD degrees in Electrical Engineering from Stanford University His research centered on computer music vocal acoustics and sound synthesis He served as Technical Director for the Center for Computer Research in Music and Acoustics researching the computer simulation of musical instruments and the singing voice controllers for real time music synthesis and performance and audio compression He has consulted and worked in the areas of DSP image compression music synthesis and speech processing for NeXT Media Vision and other companies Perry taught the audio section of the course in compression at SIGGRAPH 94 and 95 He also organized the course in Sound for Computer Graphics at SIGGRAPH 96 He is currently Assistant Professor of Computer Science with a joint appointment in Music at Princeton University researching human computer interfaces for the control of sound and music sound synthesis auditory display and immersive sound environments

Table of Contents

I Course Notes

II ANSI C Source Code

Introduction to Audio Compression and Representation

Perry R Cook

Princeton Computer Science

(also Music)

Audio Compression Overview

- *Compression in General*
- *Waveform Sampling, Storage, etc*
- *Limits of Human Audio Perception*
- *Sound and Music Representation*
- *Audio Compression Techniques*
- *Two Contrasting Compressors*
- *References and Resources*

Compression in General: Why Compress?

So Many Bits, So Little Time (Space)

- CD audio rate 2 * 2 * 8 * 44100 = 1,411,200 bps
- CD audio storage 10,584,000 bytes / minute
- A CD holds only about 70 minutes of audio
- An ISDN line can only carry 128,000 bps

Security *Best compressor removes all that is recognizable about the original sound*

Graphics people eat up all the space

Compression in General

Classical Data Compression View.

Take advantage of

- Redundancy/Correlation
- Statistics (Local / Global)
- Assumptions / Models

Problem *Much of this doesn't work directly on sound waveform data*

Waveform Sampling and Playback

- *Sample and Hold*

  Sample Rate vs Aliasing

- *Quantize*

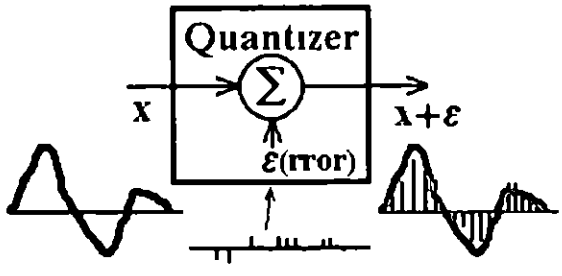
  Word Size vs Quantization Noise

- *Reconstruct Hold and Smooth (filter)*

  Filter Order vs Error and Latency

Waveform Sampling: Quantization

Quantization Introduces Noise



Examples. 16, 12, 8, 6, 4 bit music

16, 12, 8, 6, 4 bit speech

Audio Compression

Limits of Human Perception

- Time, Frequency, Amplitude, Masking, etc

Survey of Audio Compression Techniques

- Perception-Based Compression
- Production-Based Compression
- (Event-Based Compression)

Two Specific Compression Algorithms

- Production Model-Based Speech Coder
- Frequency Transform (Subband) Coder

Views of Sound

- Sound is Perceived Perception-Based
  Psychoacoustically Motivated Compression
- Sound is Produced Production-Based
  Physics/Source Model Motivated Compression
- Music(Sound) is Performed/Published/Represented
  Event-Based Compression
- Sound is a Waveform / Statistical Distribution / etc
  (these are not very good ideas in general,
  unless we get lucky (LPC))

Psychoacoustics

Limits of Human Hearing

- Time Domain Considerations
- Frequency Domain (Spectral) Considerations
- Amplitude vs Power
- Masking in Time and Frequency Domains
- Sampling Rate and Signal Bandwidth

Limits of Human Hearing

Time and Frequency

Events longer than 0 03 seconds are resolvable *in time*
shorter events are perceived as *features in frequency*

20 Hz < Human Hearing < 20 KHz
(for those under 15 or so)

"Pitch" is PERCEPTION related to FREQUENCY
Human Pitch Resolution is about 40 - 4000 Hz

Limits of Human Hearing

Amplitude or Power???

- "Loudness" is PERCEPTION related to POWER, not AMPLITUDE
- Power is proportional to (integrated) square of signal
- Human Loudness perception range is about 120 dB, where +10 db = 10 x power = 20 x amplitude
- Waveform shape is of little consequence Energy at each frequency, and how that changes in time, is the most important feature of a sound

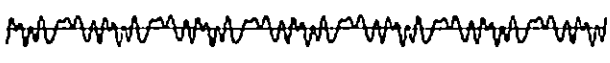
Limits of Human Hearing

Waveshape or Frequency Content??

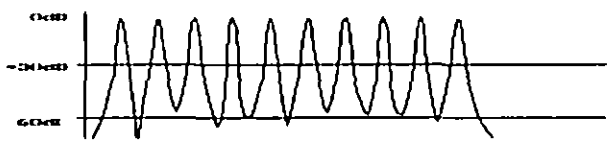
- Here are two waveforms with identical power spectra, and which are (nearly) perceptually identical

Wave 1

Wave 2



Magnitude Spectrum of Either



Limits of Human Hearing

Masking in Amplitude, Time, and Frequency

- Masking in Amplitude Loud sounds 'mask' soft ones
  Example Quantization Noise
- Masking in time A soft sound just before a louder sound is more likely to be heard than if it is just after.
  Example (and reason) Reverb vs "Preverb"
- Masking in Frequency Loud 'neighbor' frequency masks soft spectral components Low sounds mask higher ones more than high masking low

Limits of Human Hearing

Masking in Amplitude

Intuitively, a soft sound will not be heard if there is a competing loud sound Reasons

- Gain controls in the ear

  stapedes reflex and more
- Interaction (inhibition) in the cochlea
- Other mechanisms at higher levels

Limits of Human Hearing

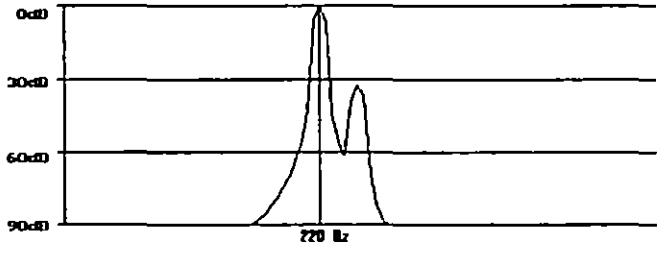
Masking in Time

- In the time range of a few milliseconds
- A soft event following a louder event tends to be grouped perceptually as part of that louder event
- If the soft event precedes the louder event, it might be heard as a separate event (become audible)

Limits of Human Hearing

Masking in Frequency

Only one component in this spectrum is audible because of frequency masking



Sampling Rates

For Cheap Compression, Look at Lowering the Sampling Rate First

44 1kHz 16 bit = CD Quality

8kHz 8 bit MuLaw = Phone Quality

Examples

Music 44 1, 32, 22 05, 16, 11 025kHz

Speech 44 1, 32, 22 05, 16, 11 025, 8kHz

Views of Sound (revisited)

Two (mainstream) views of sound and their implications for compression

1) Sound is *Perceived*

The auditory system doesn't hear everything present

- Bandwidth is limited
- Time resolution is limited
- Masking in all domains

2) Sound is *Produced*

- "Perfect" model could provide perfect compression

Perceptual Models

Exploit masking, etc , to discard perceptually irrelevant information

- Example Quantize soft sounds more accurately, loud sounds less accurately

Benefits *Generic, does not require assumptions about what produced the sound*

Drawbacks *Highest compression is difficult to achieve*

Production Models

Build a model of the sound production system, then fit the parameters

- Example If signal is speech, then a well-parameterized vocal model can yield highest quality and compression ratio

Benefits *Highest possible compression*

Drawbacks *Signal source(s) must be assumed, known, or identified*

MIDI and Other 'Event' Models

Musical Instrument Digital Interface

Represents Music as Notes and Events

and uses a synthesis engine to "render" it

An Edit Decision List (EDL) is another example

A history of source materials, transformations, and processing steps is kept Operations can be undone or recreated easily Intermediate non-parametric files are not saved

Event Based Compression

MIDI and Other Scorefiles

- A Musical Score is a very compact representation of music
- Even the score itself can be compressed further

Benefits *Highest possible compression*

Drawbacks *Cannot guarantee the "performance"*

Cannot assure the quality of the sounds

Cannot make arbitrary sounds

Event Based Compression

Enter General MIDI

- Guarantees a base set of instrument sounds,
- and a means for addressing them,
- but doesn't guarantee any quality

Better Yet, Downloadable Sounds

- Download samples for instruments
- *Benefits* *Does more to guarantee quality*
- *Drawbacks* *Samples aren't reality*

Event Based Compression

Downloadable Algorithms

- Specify the algorithm,
  the synthesis engine runs it,
  and we just send parameter changes
- Part of "Structured Audio" (MPEG4)

Benefits *Can upgrade algorithms later*
Can implement scalable synthesis

Drawbacks *Different algorithm for each class of sounds*
(but can always fall back on samples)

Back to Waveforms

Time Domain Waveform Compression

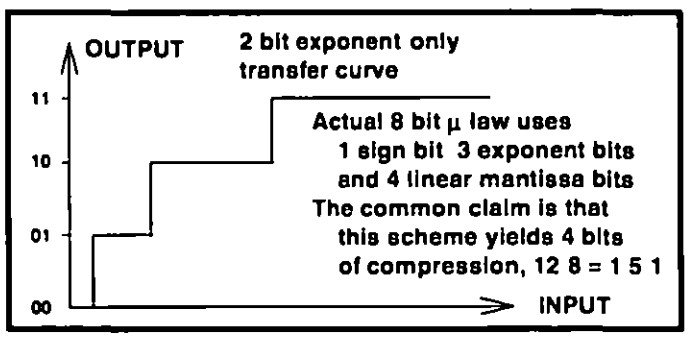
- μ – Law Non-linear amplitude quantization

- ADPCM Adaptive quantization level of changes (deltas) in signal

Time Domain Log Amplitude

μ/a-Law *More accuracy in low amplitudes, less in higher amplitudes*

Decreases perceived quantization noise



Adaptive Resolution: ADPCM

Like Log-Compressor, but bit resolution changes as a result of recent signal history

Signal differences are compressed rather than signal values

Adapting the differences (deltas) yields Adaptive Delta PCM coding, claimed to do in 4 bits what μ-law does in 8

The Frequency Domain

Exploit spectral properties to

1) Remove redundancy in signal
   - *slowly varying nature of real-world signals*
   - *periodic nature of many signals*

2) "Manage" error so it is less perceptible

Transform (Subband) Coders

Split signal into frequency subbands, then allocate bits to regions adaptively

Lossless (variable bit rate & comp ratio)

- Subbands use lower sampling rate (no advantage)
- Bands with less information use less bits
- Adaptive prediction inter/intra bands

Lossy (fixed rate and ratio)

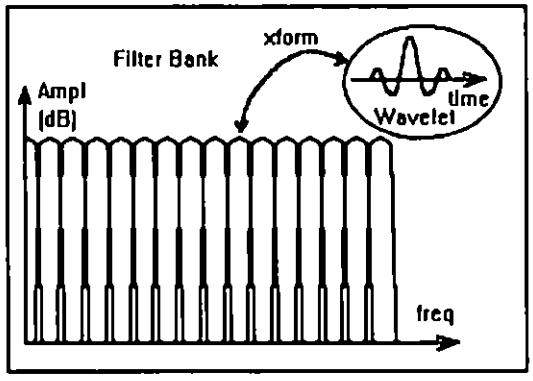
- Fix bit rate, then put bits where ear is most sensitive

Transform (Subband) Coders

Filter Bank Decomposition And Processing Can be Performed in the

Frequency Domain (FFT, etc) and/or

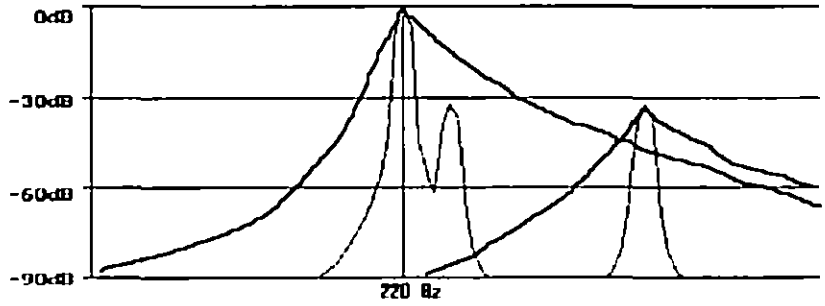
Time Domain (FIR Filterbank, Wavelets, etc)



Transform coders

Can reduce perceived quantization noise.

- frequency domain information, plus
- frequency masking knowledge



Production Models

Build a parametric model of the production system, then either

Fit the parameters to a given signal

Use signal processing techniques to extract parameters

Drive the parameters directly (no encoder?)

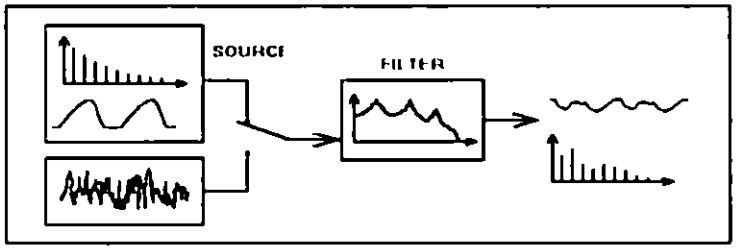
Examples Rule system to drive speech synthesizer

MIDI file to drive music synthesizer

Speech Coders (production)

Assume speech is produced by a source-filter system (vocal folds/noise + vocal tract tube)

Identify filter, type of source, then code parameters



Takes advantage of slowly varying nature of vocal tract shape and other speech parameters

Future: Multi-Model Parametric Compressors?

Analysis front end identifies source(s)

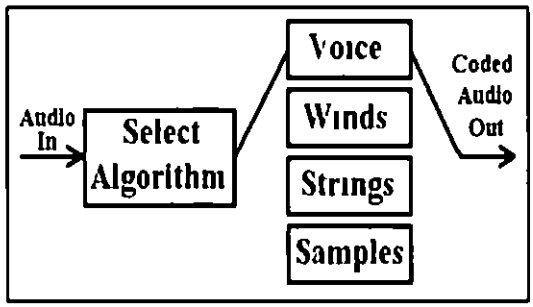
Audio is (separated and) sent to optimal model(s)

Benefits

- *High compression*
- *Other knowledge*

Drawbacks

- *We don't know how to do all this yet*



Two Contrasting Compressors

A simple speech coder

- Assume input is 8kHz, 16 bit
- 18 5 1 Ratio
- 7000 bps

A simple transform coder

- *Assume input is 22kHz, 16 bit*
- *2 (or 4) 1 Ratio*
- *176,400 (or 88200) bps*

An LPC Speech Coder

Ten pole Linear Predictive speech Coder

- Frame rate is 30 frames / second (@ 8K sampling rate)
- Frame size is 30 ms
- Source is encoded as pulse train or white noise
- LPC coefficients quantized to 2 bytes each (20 total)
- Source type coded in 1 bit (pitched/noise) per frame
- Source amplitude stored in one float per frame
- Source pitch stored in one float per frame
- Total transmission rate 7000 bps (18 5 1 ratio)

A Cheap Transform Coder

FHT-based Delta Block Adaptive Log Amplitude Transform Coder

- 64 point (32 subbands) FHT Frame (3 ms @ 22kHz)
- Frame rate is 344 frames/second
- Deltas of signal are used
- 4 (or 8) bit logarithmic compression of each band
- Each block peak is detected and stored as a short int
- Compression is 2 (or 4) 1 (plus silence)

References and Resources

General Psychoacoustics Books

Bregman, *Auditory Scene Analysis*, MIT Press, 1990

Dowling and Harwood, *Music Cognition*, Academic Press, 1986

Handel, *Listening an Introduction to the Perception of Auditory Events*, MIT, Cambridge, MA, 1989

McAdams and Bigand (eds), *Thinking in Sound the Cognitive Psychology of Human Audition*, Oxford Univ Press, NY, 1993

Pierce, *The Science of Musical Sound*, Freeman, New York, 1992

Roederer, *Introduction to the Physics and Psychophysics of Music*, Springer-Verlag, New York, 1975

References and Resources

Critical Bands and Masking

Old Views

Zwicker, Flottorp, and Stevens, "Critical Bandwidth in Loudness Summation", J Acoustical Soc America 29, 1957

Newer Views

Moore and Glasberg, "Suggested Formulae for Calculating Auditory-Filter Bandwidths and Excitation Patterns," JASA, 7, 4(3) 1983

References and Resources

Mu-Law, ADPCM Coding

Smith, "Instantaneous Companding of Quantized Signals," Bell Systems Tech Journal, Vol 36, No 3, May 1957

IMA Compatibility Proceedings, Section 6, "ADPCM," May 1992

Chaffan, "High Quality Speech Synthesis Using ADPDM Technology," SAE Technical Paper Series #831023, 1983

Pohlman, "Principles of Digital Audio," Sams Books, 1993

References and Resources

Speech Models and Compression

Makhoul Linear Prediction, a Tutorial Review," Proceedings of the IEEE, V 63, pp 560-580, 1975

Spanias, "Speech Coding, a Tutorial Review, Proc IEEE, 82 10, 1994,

Rabiner and Schafer, *Digital Processing of Speech Signals*, Prentice Hall, 1978

O Shaughnessy, *Speech Communication, Human and Machine*, Addison Wesley, 1987

References and Resources

Subband Coding, Wavelets, AC-2

Tribolet and Crochiere, "Frequency Domain Coding of Speech," IEEE ASSP 27 5, 1979

Rioul and Vetterli, "Wavelets and Signal Processing, ' IEEE Signal Processing Magazine, 1991

Davidson, Anderson, and Lovrich, A Low-Cost Adaptive Transform Decoder Implementation for High-Quality Audio," (AC-2) IEEE Pub 0-7803-0532 9/92, 1992

References and Resources

MPEG

Dehery, Lever and Urcun A MUSICAM Source CODEC for Digital Audio Broadcasting and Storage, ICASSP A1 9 1991

Stoll, Theile, and Link, MASCAM Using Psychoacoustic Masking Effects for Low Bit Rate Coding of High Quality Complex Sounds, 84th AES, Paris, 1988

Stoll and Dehery, MUSICAM High Quality Audio Bit Rate Reduction System Family for Different Applications, IEEE Conf on Communications 1990.

ISO/IEC Working Papers & Standards Reports Example JTCI SC29 WG11 N0403 MPEG 93/479, 1993

Brandenburg and Bosi, Overview of MPEG Audio Current and Future Standards for Low-Bit Rate Audio Coding Journal of the AES, 45 1/2 1997

MIDI and Music Representation

The Complete MIDI 1 0 Detailed Specification, MIDI Manufacturers Association, La Habra, CA, MMA, 1996

Jungleib, *General MIDI,* A-R Editions, 1995

Selfridge-Field, *Beyond MIDI, The Handbook of Musical Codes,* MIT Press, 1997

Grill, Edler, Kaneko, Lee, Nishiguchi, Scheirer, and Väänänen (eds), ISO 14496-3 (MPEG-4 Audio), Committee Draft, ISO/IEC JTCI/SC29/WG11, document W1903, Fribourg CH, October 1997

Wright, White, Fay, and Petkevich, "The Downloadable Sounds Level 1 Specification," Proceedings of the International Computer Music Conference, 1997

Source Code

Quantization Program (N bit)

MuLaw Coder/Decoder (8 Bit)

SigLaw Coder/Decoder (4 bit)

ADPCM Coder/Decoder (4 bit)

Xform Coder/Decoder (4 and 8 bit)

LPC Speech Coder/Decoder

Utilities

// readme.txt

```
/*************************************************************/
/***    Compression and utility source code for            ***/
/***    SIGGRAPH 1998 audio compression course             ***/
/***        by Perry R  Cook, Princeton University         ***/
/***                                                       ***/
/***    These all operate on raw 16 bit mono files         ***/
/***    or the compressed versions of same                 ***/
/***                                                       ***/
/*************************************************************/

quantize c  //  Quantizes file to the number of specified bits

siglaw4 c   //  4-bit exponent/sign compressor
desigla4 c  //  decompressor for above

mulaw8 c    //  8-bit mu-law log compressor
demulaw8 c  //  decompressor for above

acpdmcod c  //  4-bit adaptive delta compressor
adpcmdec c  //  decompressor for above

sigxfor8 c  //  8-bit block adaptive transform coder
sigxinv8 c  //  decompressor for above
sigxfor4 c  //  4-bit block adaptive transform coder
sigxinv4 c  //  decompressor for above
fhtrx4 c    //  Hartley Transform code used by above

fitlpc c    //  Linear Predictive speech Coder
lpcresyn c  //  resynthesis model for above

sines c     //  make some sine waves for testing
byteswap c  //  swap bytes for Intel/Motorola (other/other)
diffrnce c  //  form difference between two files


Source Sound Files:

  \sounds\speech raw   //  test speech at 8kHz, 16 bit
  \sounds\music raw    //  test music at 22kHz, 16 bit
```

// quantize.c

```
/*******************************************************/
/***    Quantization demonstration code for          ***/
/***    SIGGRAPH 1998 audio compression course       ***/
/***        by Perry R  Cook, Princeton University   ***/
/***    This program takes a 16 bit linear signed    ***/
/***    headerless file and quantizes it to the      ***/
/***    specified number of bits   It does not       ***/
/***    compress the file, but just quantizes it     ***/
/***    for quality comparison purposes              ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

void err_msg()
{
    fprintf(stderr,"useage: quantize numBits inputFile outputFile\n\n");
}

void main(int argc, char *argv[]) {
    FILE *file_in,*file_out;
    int num_bits;
    short data;
    float temp,temp2;

    if (argc==4)        {
      file_in = fopen(argv[2],"rb");
      file_out = fopen(argv[3],"wb");
      num_bits = atoi(argv[1]);
      num_bits = 16 - num_bits;
      temp = pow(2 0,num_bits);
      if (file_in && file_out) {
          while(fread(&data,2,1,file_in)) {
            temp2 = (float) data / temp;
                data = temp2;
            data *= temp;
                fwrite(&data,2,1,file_out);
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
      err_msg();
      exit(0);
    }
    exit(0);
}
```

// siglaw4.c

```
/*******************************************************/
/***    4 bit log compression code for               ***/
/***    SIGGRAPH 1998 audio compression course       ***/
/***       by Perry R  Cook, Princeton University    ***/
/***    This program takes a 16 bit linear signed    ***/
/***    headerless file and compresses it 4:1        ***/
/***    Use desigla4 c to decompress files           ***/
/***    The scheme here is exponent-only encoding    ***/
/***    which would be really cheap in hardware      ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    unsigned short data_out;
    short i,data_in[4] = {0 0};
    short abslut;
    float exponent;
    short Exp=0,Sign=0;

    if (argc==3)        {

      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
      if (file_in && file_out){
          while(fread(&data_in,2,4,file_in)) {
            data_out = 0;
            for (i=0;i<4;i++)     {
                abslut = abs(data_in[i]);            // Absolute of data
                Sign = (data_in[i]<0);                 // Sign of data
                if (abslut<256)   {
                  Exp = 0;
                }
                else      {
                  exponent = log(abslut)/log(2 0);   // Log base 2
                  Exp = exponent - 8;                // Like division
                                                     //  in linear space
                }
                data_out = data_out + (Exp<<(i*4));    // Pack nybbles
                if (Sign)       {                      // Set sign bit
                  if (i==0)
                      data_out = data_out | 8;
                  else if (i==1)
                      data_out = data_out | 128;
                  else if (i==2)
                      data_out = data_out | 2048;
                  else if (i==3)
                      data_out = data_out | 0x8000;
                }
            }
            fwrite(&data_out,2,1,file_out);    // write out in sets of 4
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
        fprintf(stderr,"format is: siglaw4 inputFile outputFile\n");
    }
    exit(0);
}
```

// desigla4.c

```
/*************************************************************/
/***    4 bit log decompression code for                   ***/
/***    SIGGRAPH 1998 audio compression course             ***/
/***       by Perry R  Cook, Princeton University          ***/
/***    This program converts files created                ***/
/***    with siglaw4 c back into 16 bit linear             ***/
/***    The scheme here is exponent-only encoding          ***/
/***    which would be really cheap in hardware            ***/
/*************************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    unsigned short data_in;
    int i,
    short data_out[4] = {0 0};
    short nib_data[4];
    short Exp=0,Sign=0;

    if (argc==3)        {
      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
      if (file_in && file_out){
          while(fread(&data_in,2,1,file_in)) {
                nib_data[0] = data_in & 15;            // Unpack nybbles
            nib_data[1] = (data_in >> 4) & 15;
            nib_data[2] = (data_in >> 8) & 15;
            nib_data[3] = (data_in >> 12) & 15,
            for (i=0;i<4;i++)       {
                    Exp = nib_data[i] & 7;             // Get exponent
                    if (nib_data[i] & 8)              // Then check sign
              Sign = 1;
                else
                  Sign = 0;
                if (!Sign)   {
                  if      (Exp==0)
                      data_out[i] = 0;
                  else
                      data_out[i] = 255 * pow(2,Exp);
                }
                    else        {                   // 2's compliment junk
                    if      (Exp==0)
                      data_out[i] = 127 * 255;
                  else if (Exp==1)
                      data_out[i] = 126 * 255,
                  else if (Exp==2)
                      data_out[i] = 124 * 255;
                  else if (Exp==3)
                      data_out[i] = 120 * 255;
                  else if (Exp==4)
                      data_out[i] = 112 * 255;
                  else if (Exp==5)
                      data_out[i] = 96 * 255;
                  else if (Exp==6)
                      data_out[i] = 64 * 255,
                  else if (Exp==7)
                      data_out[i] = 0;
                        data_out[i] = data_out[i] | -32768; // Sign bit
                }
              }
              fwrite(data_out,2,4,file_out);
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
        fprintf(stderr,"format is  desigla4 inputFile outputFile\n");
    }
    exit(0);
}
```

// mulaw8.c

```
/*******************************************************/
/***    8 bit log compression code for                ***/
/***    SIGGRAPH 1998 audio compression course        ***/
/***       by Perry R  Cook, Princeton University     ***/
/***    This program takes a 16 bit linear signed     ***/
/***    headerless file and compresses it 2 1         ***/
/***    Use demulaw8 c to decompress files            ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    unsigned short data_out,
    short i,data_in[2];
    unsigned short abslut,
    float temp,scale,mu = 0 062;
    short Sign=0;

    if (argc==3)        {

      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
        scale = log(1 0 + mu);
        if (file_in && file_out){
            while(fread(&data_in,2,2,file_in)) {     // Load two words
                abslut = abs(data_in[0]);            // Absolute of data
                Sign = (data_in[0]<0);               // Sign of data
                temp = log(1 0 + (mu * (float) abslut)); // Basic Mulaw
                temp /= scale,                          //   equation
                data_out = (unsigned short) temp;    // Pack first byte
                if (Sign) data_out = data_out | 128; // with sign bit

                abslut = abs(data_in[1]);            // Absolute of data
                Sign = (data_in[1]<0);               // Sign of data
                temp = log(1 0 + (mu * (float) abslut)); // Basic Mulaw
                temp /= scale;                          //   equation
                data_out = (data_out << 8)           // Pack
                        + (unsigned short) temp;     //  second byte
                if (Sign) data_out = data_out | 128; //    with sign bit

                fwrite(&data_out,2,1,file_out);   // write out sets of 4
            }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out),
    }
    else {
        fprintf(stderr,"format is  mulaw8 inputFile outputFile\n");
    }
    exit(0);
}
```

// demulaw8.c

```
/*******************************************************/
/***    8 bit log decompression code for             ***/
/***    SIGGRAPH 1998 audio compression course       ***/
/***        by Perry R  Cook, Princeton University   ***/
/***    This program converts files created          ***/
/***    with mulaw8 c back into 16 bit linear        ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    unsigned short data_in;
    unsigned char char_data;
    int i;
    short data_out[2] = {0 0};
    short Sign=0;
    float temp,scale,mu = 0 062;

    if (argc==3)        {
      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
      if (file_in && file_out){
            scale = log(1 0 + mu);
            while(fread(&data_in,2,1,file_in)) {  // Read compressed
                                                  //      bytes
                char_data = data_in >> 8,         // Get first byte
                if (char_data & 128)              // Then check sign
                    Sign = 1;
                else
                    Sign = 0;
                char_data = char_data & 127;      // Absolute data
                temp = (float) char_data * scale; // Undo mulaw
                temp = (exp(temp) - 1 0) / mu;    //   compression
                if (Sign) temp = -temp;           //     and add sign
                data_out[0] = temp;

                char_data = data_in & 255;        // Get second byte
                if (char_data & 128)              // Then check sign
                    Sign = 1;
                else
                    Sign = 0;
                char_data = char_data & 127;      // Absolute data
                temp = (float) char_data * scale; // Undo mulaw
                temp = (exp(temp) - 1 0) / mu;    //   compression
                if (Sign) temp = -temp,           //     and add sign
                data_out[1] = temp,

                fwrite(data_out,2,2,file_out);
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
        fprintf(stderr,"format is: demulaw8 inputFile outputFile\n");
    }
    exit(0);
}
```

// acpcmcod.c

```
/*********************************************************/
/***     4 bit ADPCM compression code for              ***/
/***     SIGGRAPH 1998 audio compression course        ***/
/***         by Perry R  Cook, Princeton University    ***/
/***     This program takes a 16 bit linear signed     ***/
/***     headerless file and compresses it 4:1         ***/
/***     Use adpcmdec c to decompress files            ***/
/*********************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

#define MAX_STEP 2048
#define MIN_STEP 16

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    long i;
    short XHAT1=0,Xn,Dn;
    short deln,del1=8;
    short delndec,del1dec=8,ln1dec;
    short ln1=1,sign,temp;
    short data[4];
    unsigned short data_out,
    float M[16] =
{0 909,0 909,0 909,0 909,1 21,1 4641,1 771561,2 143589,
              0 909,0 909,0 909,0 909,1 21,1 4641,1 771561,2 143589};
    float del_table[16] = {0 0,0 25,0 5,0 75,1 0,1 25,1 5,1 75,
                  0 0,-0 25,-0 5,-0 75,-1 0,-1 25,-1 5,-1 75};

    if (argc==3)        {
      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
      if (file_in && file_out) {
          while(fread(&data,2,4,file_in)) {
            data_out = 0;
            for (i=0;i<4;i++)       {
                Xn = data[i];
                Dn = Xn - XHAT1;            //   Form delta signal

                deln = del1 * M[ln1];       //   Get new delta
                temp = deln - MIN_STEP;     //   delta < min ???
                if (temp<0)    {            //   Skip 1 on positive
                  deln = MIN_STEP;          //   set it to min
                }
                temp = MAX_STEP - deln;     //   delta > max ???
                if (temp<0) {               //   Skip 1 on positive
                  deln = MAX_STEP;          //   set it to max
                }
                del1 = deln;                //   Stash delta

                ln1 = 0;

                temp = Dn;                  //   to set flags
                if (temp<0)  {              //   Skip on positive
```

```
                lnl = 8;                    //   Else set negative
                Dn = -Dn;                   //      bit in code word
              }
              temp = deln - Dn;             //   Check magnitude
              if (temp<0)  {                //   Skip 2 on positive
                lnl += 4;                   //   set MSB in code word
                Dn -= deln;                 //   decrease magnitude
              }
              temp = deln*0 5 - Dn;         //   Check magnitude
              if (temp<0)       {           //   Skip 2 on positive
                lnl += 2;                   //   set 2SB in code word
                Dn -= deln*0 5;             //   decrease magnitude
              }
              temp = deln*0 25 - Dn;        //   Check magnitude
              if (temp<0)   {               //   Skip 2 on positive
                lnl += 1,                   //   set LSB in code word
              }

/******************  DECODER EMULATION  *********************/

              lnldec = lnl,

              delndec = delldec * M[lnldec];    //  form new delta
              temp = delndec - MIN_STEP,        //  test against min
              if (temp<0)                       //  Skip 1 on positive
                delndec = MIN_STEP;             //  set to min
              temp = MAX_STEP - delndec;        //  test against max
              if (temp<0)                       //  Skip 1 on positive
                delndec = MAX_STEP;             //  set to max
              delldec = delndec;                //  save delta

              XHAT1 += delndec * del_table[lnldec], // update sample

              if (XHAT1>32000 || XHAT1<-32000) //  Skip if no saturate
                XHAT1 *= 0 95;                  //  Else fix it

/*************************************************************/

                data_out = (data_out << 4) + lnl;
            }
            fwrite(&data_out,2,1,file_out);
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
      fprintf(stderr,"format is: adpcmcod inputFile outputFile\n\n");
      exit(0);
    }
    exit(0);
}
```

// adpcmdec.c

```
/*******************************************************/
/***     4 bit ADPCM decompression code for          ***/
/***     SIGGRAPH 1998 audio compression course      ***/
/***        by Perry R  Cook, Princeton University   ***/
/***     This program decompresses files created     ***/
/***     by adpcmcod c                               ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <math h>
#include <string h>

#define MAX_STEP 2048
#define MIN_STEP 16

void main(short argc, char *argv[]) {
    FILE *file_in,*file_out;
    long i,
    short XHAT1=0,Xn,Dn;
    short deln;
    short delndec,del1dec=8,ln1dec;
    short sign,temp,
    short data[4],
    unsigned short data_in,
    float M[16] =
{0 909,0 909,0 909,0 909,1 21,1 4641,1 771561,2 143589,
              0 909,0 909,0 909,0 909,1 21,1 4641,1 771561,2 143589};
    float del_table[16] = {0 0,0 25,0 5,0 75,1 0,1 25,1 5,1 75,
                  0 0,-0 25,-0 5,-0 75,-1 0,-1 25,-1 5,-1 75},

    if (argc==3)        {
      file_in = fopen(argv[1],"rb");
      file_out = fopen(argv[2],"wb");
      if (file_in && file_out) {
          while(fread(&data_in,2,1,file_in)) {
            for (i=0;i<4;i++)       {
                ln1dec = data_in >> 12;          //  Load data
                data_in = data_in << 4,

                delndec = del1dec * M[ln1dec];   //  form new delta
                temp = delndec - MIN_STEP;       //  test against min
                if (temp<0)                      //  Skip 1 on positive
                  delndec = MIN_STEP,            //  set to min
                temp = MAX_STEP - delndec;       //  test against max
                if (temp<0)                      //  Skip 1 on positive
                  delndec = MAX_STEP,            //  set to max
                del1dec = delndec;               //  save delta

                XHAT1 += delndec * del_table[ln1dec]; //  update sample

                if (XHAT1>32000 || XHAT1<-32000) //  Skip if no saturate
                  XHAT1 *= 0 95;                 //  Else fix it

                data[i] = XHAT1;                 //  Output Samples

            }
            fwrite(&data,2,4,file_out),
          }
      }
      else {
          fprintf(stderr,"File troubles   Check names, paths, etc \n");
      }
      fclose(file_in);
      fclose(file_out);
    }
    else {
      fprintf(stderr,"format is  adpcmdec inputFile outputFile\n\n");
      exit(0);
    }
    exit(0);
}
```

// sigxfor8.c

```
/*********************************************************/
/***    Simple 8 bit Transform compression code for    ***/
/***    SIGGRAPH 1998 audio compression course         ***/
/***        by Perry R  Cook, Princeton University     ***/
/***    This program takes a 16 bit linear signed      ***/
/***    headerless file and compresses it 2 1          ***/
/***    Use sigxinv8 c to decompress files             ***/
/*********************************************************/

#include <stdlib h>
#include <stdio h>
#include <math h>
#include <assert h>

#include "fhtrx4 c"

#define FHT_LENGTH 64
#define POWER_OF_FOUR 3

main(int argc,char *argv[])
{
    FILE *fileIn,*fileOut,
    short max, x_input[FHT_LENGTH];
    float fmax, temp_float[FHT_LENGTH];
    signed char y_out[FHT_LENGTH];
    int i, n_read;
    extern void fhtRX4(int powerOfFour, float *array);

    if (argc!=3) {
        fprintf(stderr,"usage  %s filein fileout\n",argv[0]);
      exit(0);
    }
    fileIn = fopen(argv[1],"rb");
    if (!fileIn)   {
        printf("Input file problem\n"),
        fclose(fileIn);
        exit(0),
    }
    fileOut = fopen(argv[2],"wb");
    if (!fileOut)       {
        printf("Output file problem\n");
      fclose(fileIn);
      exit(0);
    }

    n_read = fread(x_input,2,FHT_LENGTH,fileIn);
    while (n_read)    {
        for (i=n_read;i<FHT_LENGTH;i++)                  // End Of File
            x_input[i] = 0;                              //     condition

        temp_float[0] = x_input[0];                      // first sample
        max = abs(x_input[0]);
        for (i=1;i<FHT_LENGTH;i++)        {              // compute block
            if (max < abs(x_input[i]))                   // dynamic range
                max = abs(x_input[i]);
            temp_float[i] = x_input[i] - x_input[i-1];   // sample deltas
        }
```

```
        if (max > 8)    {
            fwrite(&max,2,1,fileOut);                   // write block max
            fhtRX4(POWER_OF_FOUR,temp_float);           // do transform
            fmax = fabs(temp_float[0]);
            for (i=1;i<FHT_LENGTH;i++)          {       // compute transform
                if (fmax < fabs(temp_float[i]))         // maximum for code
                    fmax = fabs(temp_float[i]);         //    normalization
            }
            for (i=0,i<FHT_LENGTH;i++) {                // pack data
                y_out[i]  = 127 * temp_float[i] / fmax;
            }
            fwrite(y_out,FHT_LENGTH,1,fileOut);
        }
        else {
            printf("Zero Block Here\n");
            max = 0;                                    // Special silence
            fwrite(&max,2,1,fileOut);                   //   write block max
        }
        n_read = fread(x_input,2,FHT_LENGTH,fileIn);
    }

    fclose(fileIn);
    fclose(fileOut);
    return;
}
```

// sigxinv8.c

```
/*************************************************************/
/***    Simple 8 bit Transform decompression code          ***/
/***    for SIGGRAPH 1998 audio compression course         ***/
/***       by Perry R  Cook, Princeton University          ***/
/***    This program eats the nybble packed data           ***/
/***    generated by sigxfor8 c and decompresses           ***/
/***    into a 16 bit linear signed headerless file        ***/
/*************************************************************/

#include <stdlib h>
#include <stdio h>
#include <math h>
#include <assert h>

#include "fhtrx4 c"

#define FHT_LENGTH 64
#define POWER_OF_FOUR 3

main(int argc,char *argv[])
{
    FILE *fileIn,*fileOut;
    short x_out[FHT_LENGTH];
    float temp_float[FHT_LENGTH];
    signed char y_input[FHT_LENGTH];
    float fmax,temp;
    short max,first_sample;
    int n_read;
    long i;
    extern void fhtRX4(int powerOfFour, float *array),

    if (argc!=3) {
        fprintf(stderr,"usage  %s filein fileout\n",argv[0]);
      exit(0);
    }
    fileIn = fopen(argv[1],"rb");
    if (!fileIn)    {
        printf("Input file problem\n");
        fclose(fileIn),
        exit(0);
    }
    fileOut = fopen(argv[2],"wb"),
    if (!fileOut)       {
        printf("Output file problem\n");
      fclose(fileIn);
      exit(0);
    }

    n_read = fread(&max,2,1,fileIn);
    while (n_read)    {
        if (max)    {
            n_read = fread(y_input,1,FHT_LENGTH,fileIn);
            for (i=0;i<FHT_LENGTH;i++)
                temp_float[i] = y_input[i];
            fhtRX4(POWER_OF_FOUR,temp_float);    //  Back to time domain
            fmax = temp_float[0];
            for (i=1;i<FHT_LENGTH;i++)  {
                temp_float[i] += temp_float[i-1], // integrate back
                if (fmax < fabs(temp_float[i]))   // and remember peak
                    fmax = fabs(temp_float[i]);   // for renormalization
            }
            for (i=0,i<FHT_LENGTH;i++)      {
                x_out[i] = max * temp_float[i] / fmax; // renormalize
            }
        }
        else    {                                  //  Special Silence Code
            for (i=0;i<FHT_LENGTH;i++)
                x_out[i] = 0;
        }
        n_read = fread(&max,2,1,fileIn);
        fwrite(x_out,FHT_LENGTH,2,fileOut);
    }

    fclose(fileIn);
    fclose(fileOut),
    return;
}
```

// sigxfor4.c

```
/*********************************************************/
/***    Simple 4 bit Transform compression code for    ***/
/***    SIGGRAPH 1998 audio compression course         ***/
/***        by Perry R  Cook, Princeton University     ***/
/***    This program takes a 16 bit linear signed      ***/
/***    headerless file and compresses it 4 1          ***/
/***    Use sigxinv4 c to decompress files             ***/
/*********************************************************/

#include <stdlib h>
#include <stdio h>
#include <math h>
#include <assert h>

#include "fhtrx4 c"

#define FHT_LENGTH 64
#define POWER_OF_FOUR 3

main(int argc,char *argv[])
{
    FILE *fileIn,*fileOut;
    short max, temp_word, x_input[FHT_LENGTH];
    float fmax, temp_float[FHT_LENGTH];
    signed char y_out[FHT_LENGTH];
    int abslut,Exp,Sign;
    int i,n_read;
    extern void fhtRX4(int powerOfFour, float *array);

    if (argc!=3) {
        fprintf(stderr,"usage: %s filein fileout\n",argv[0]),
      exit(0);
    }
    fileIn = fopen(argv[1],"rb");
    if (!fileIn)   {
        printf("Input file problem\n");
        fclose(fileIn);
        exit(0),
    }
    fileOut = fopen(argv[2],"wb");
    if (!fileOut)       {
        printf("Output file problem\n"),
      fclose(fileIn);
      exit(0);
    }

    n_read = fread(x_input,2,FHT_LENGTH,fileIn);
    while (n_read)    {
        for (i=n_read;i<FHT_LENGTH;i++)              // End Of File
            x_input[i] = 0;                          //     condition

        temp_float[0] = x_input[0];                  // store 1st sample
        max = abs(x_input[0]);
        for (i=1;i<FHT_LENGTH;i++)         {         // compute block
            if (max < abs(x_input[i]))               // dynamic range
                max = abs(x_input[i]);
            temp_float[i] = x_input[i] - x_input[i-1];  // sample deltas
```

```
        }
        fhtRX4(POWER_OF_FOUR,temp_float);           // do transform
        fmax = fabs(temp_float[0]);

        if (max > 8) {
            fwrite(&max,2,1,fileOut);               // write block max
            for (i=1,i<FHT_LENGTH;i++)        {     // compute transform
                if (fmax < fabs(temp_float[i]))     // maximum for
                    fmax = fabs(temp_float[i]);     //   normalization
            }

            for (i=0;i<FHT_LENGTH/2,i++) {          // pack data
                temp_word = 255 * temp_float[2*i] / fmax;
                abslut = abs(temp_word),            // Absolute of data
                Sign = (temp_word<0);               // Sign of data
                if (abslut < 1)
                    Exp = 0;
                else
                    Exp = log(abslut)/log(2 0);     // Log base 2
                y_out[i] = Exp << 4;
                if (Sign)                           // Set sign bit
                    y_out[i] = y_out[i] | 128;

                temp_word = 255 * temp_float[2*i+1] / fmax;
                abslut = abs(temp_word);            // Absolute of data
                Sign = (temp_word<0);               // Sign of data
                if (abslut < 1)
                Exp = 0,
                else
                    Exp = log(abslut)/log(2 0);     // Log base 2
                y_out[i] += Exp;
                if (Sign)                           // Set sign bit
                    y_out[i] = y_out[i] | 8;
            }
        }
        else    {
            max = 0;                                // Special zero
            fwrite(&max,2,1,fileOut);               //   write block max
        }

        n_read = fread(x_input,2,FHT_LENGTH,fileIn);
        fwrite(y_out,FHT_LENGTH/2,1,fileOut);
    }

    fclose(fileIn);
    fclose(fileOut);
    return,
}
```

// sigx1nv4.c

```
/*********************************************************/
/***    Simple 4 bit Transform decompression code      ***/
/***    for SIGGRAPH 1998 audio compression course     ***/
/***        by Perry R  Cook, Princeton University     ***/
/***    This program eats the nybble packed data       ***/
/***    generated by sigxfor4 c and decompresses it    ***/
/***    into a 16 bit linear signed headerless file    ***/
/*********************************************************/

#include <stdlib h>
#include <stdio h>
#include <math h>
#include <assert h>

#include "fhtrx4 c"

#define FHT_LENGTH 64
#define POWER_OF_FOUR 3

main(int argc,char *argv[])
{
    FILE *fileIn,*fileOut;
    short max, x_out[FHT_LENGTH];
    float fmax, temp_float[FHT_LENGTH];
    signed char y_input[FHT_LENGTH];
    unsigned char temp_byte;
    short Exp=0,Sign=0;
    int i, n_read;
    extern void fhtRX4(int powerOfFour, float *array);

    if (argc!=3) {
        fprintf(stderr,"usage  %s filein fileout\n",argv[0]);
      exit(0);
    }
    fileIn = fopen(argv[1],"rb"),
    if (!fileIn)   {
        printf("Input file problem\n"),
        fclose(fileIn);
        exit(0);
    }
    fileOut = fopen(argv[2],"wb");
    if (!fileOut)       {
        printf("Output file problem\n");
      fclose(fileIn);
      exit(0),
    }

    n_read = fread(&max,2,1,fileIn);
    while (n_read)    {
        if (max)    {
            n_read = fread(y_input,1,FHT_LENGTH/2,fileIn);
            for (i=0,i<FHT_LENGTH/2;i++)      {
                temp_byte = (y_input[i] >> 4) & 15;
                Exp = temp_byte & 7;                // Get exponent
                if (temp_byte & 8)                  // Then check sign
                    Sign = 1;
                else
```

```
                Sign = 0;

            temp_float[2*i] = pow(2,Exp);       // Set log data
            if (Sign)   {                       // and add sign
                temp_float[2*i] *= -1;          //     if needed
            }

            temp_byte = y_input[i] & 15;
            Exp = temp_byte & 7,                // Get exponent
            if (temp_byte & 8)                  // Then check sign
               Sign = 1;
            else
                Sign = 0;
            temp_float[2*i+1] = pow(2,Exp);     // Set log data
            if (Sign)   {                       // and add sign
                temp_float[2*i+1] *= -1;        //     if needed
            }
        }
        fhtRX4(POWER_OF_FOUR,temp_float);     // back to time domain
        fmax = temp_float[0];
        for (i=1;i<FHT_LENGTH;i++) {
            temp_float[i] += temp_float[i-1];  // integrate back
            if (fabs(temp_float[i]) > fmax)    // and remember peak
                fmax = fabs(temp_float[i]);   // for renormalization
        }
        for (i=0;i<FHT_LENGTH;i++)      {
            x_out[i] = max * temp_float[i] / fmax; // renormalize
        }
    }
    else    {                                 //  Special Silence Code
        for (i=0;i<FHT_LENGTH;i++)
            x_out[i] = 0;
    }
    n_read = fread(&max,2,1,fileIn),
    fwrite(x_out,FHT_LENGTH,2,fileOut);
}

fclose(fileIn),
fclose(fileOut);
return;
}
```

// fhtrx4.c

```
#define PI 3 141592654782
#define SQRT_TWO 1 414213562
#define TWOPI 6 283185309564

void fhtRX4(int powerOfFour, float *array)
{
    /*  In place Fast Hartley Transform of floating point data in array
      Size of data array must be power of four  Lots of sets of four
      inline code statements, so it is verbose and repetitive, but fast
      A 1024 point FHT takes approximately 80 milliseconds on the NeXT
      computer (not in the DSP 56001, just in compiled C as shown here)

      The Fast Hartley Transform algorithm is patented, and is
      documented in the book "The Hartley Transform", by Ronald N
      Bracewell   This routine was converted to C from a BASIC routine
      in the above book, that routine Copyright 1985, The Board of
      Trustees of Stanford University       */

    register int j=0,i=0,k=0,L=0;
    int n=0,n4=0,d1=0,d2=0,d3=0,d4=0,d5=1,d6=0,d7=0,d8=0,d9=0;
    int L1=0,L2=0,L3=0,L4=0,L5=0,L6=0,L7=0,L8=0;
    float r=0 0;
    float a1=0,a2=0,a3=0;
    float t=0 0,t1=0 0,t2=0 0,t3=0 0,t4=0 0,t5=0 0,t6=0 0,t7=0 0,t8=0 0;
    float t9=0 0,t0=0 0;
    float c1,c2,c3,s1,s2,s3;

    n = pow(4 0 , (double) powerOfFour);
    n4 = n / 4;
    r = SQRT_TWO;
    j = 1;
    i = 0;
    while (i<n-1)       {
      i++,
      if (i<j)         {
            t = array[j-1];
            array[j-1] = array[i-1];
            array[i-1] = t;
      }
      k = n4;
      while ((3*k)<j) {
            j -= 3 * k;
            k /= 4;
      }
      j += k;
    }
    for (i=0;i<n;i += 4) {
      t5 = array[i];
      t6 = array[i+1];
      t7 = array[i+2];
      t8 = array[i+3];
      t1 = t5 + t6;
      t2 = t5 - t6,
      t3 = t7 + t8;
      t4 = t7 - t8;
      array[i] = t1 + t3;
      array[i+1] = t1 - t3;
      array[i+2] = t2 + t4;
      array[i+3] = t2 - t4;
    }
    for (L=2;L<=powerOfFour;L++)  {
      d1 = pow(2 0 , L+L-3 0);
      d2=d1+d1;
      d3=d2+d2;
      d4=d2+d3;
      d5=d3+d3;
      for (j=0;j<n;j += d5)     {
```

```
            t5 = array[j];
            t6 = array[j+d2];
            t7 = array[j+d3];
            t8 = array[j+d4];
            t1 = t5+t6;
            t2 = t5-t6;
            t3 = t7+t8;
            t4 = t7-t8;
            array[j] = t1 + t3;
            array[j+d2] = t1 - t3;
            array[j+d3] = t2 + t4;
            array[j+d4] = t2 - t4;
            d6 = j+d1;
            d7 = j+d1+d2;
            d8 = j+d1+d3;
            d9 = j+d1+d4;
            t1 = array[d6];
            t2 = array[d7] * r;
            t3 = array[d8];
            t4 = array[d9] * r;
            array[d6] = t1 + t2 + t3;
            array[d7] = t1 - t3 + t4;
            array[d8] = t1 - t2 + t3;
            array[d9] = t1 - t3 - t4;
            for (k=1;k<d1;k++) {
               L1 = j + k;
               L2 = L1 + d2;
               L3 = L1 + d3;
               L4 = L1 + d4;
               L5 = j + d2 - k;
               L6 = L5 + d2;
               L7 = L5 + d3;
               L8 = L5 + d4;
               a1 = (float) k / (float) d3 * PI;
               a2 = a1 + a1;
               a3 = a1 + a2;
               c1 = cos(a1);
               c2 = cos(a2);
               c3 = cos(a3);
               s1 = sin(a1);
               s2 = sin(a2);
               s3 = sin(a3);
               t5 = array[L2] * c1 + array[L6] * s1;
               t6 = array[L3] * c2 + array[L7] * s2;
               t7 = array[L4] * c3 + array[L8] * s3;
               t8 = array[L6] * c1 - array[L2] * s1;
               t9 = array[L7] * c2 - array[L3] * s2;
               t0 = array[L8] * c3 - array[L4] * s3;
               t1 = array[L5] - t9;
               t2 = array[L5] + t9;
               t3 = - t8 - t0;
               t4 = t5 - t7;
               array[L5] = t1 + t4;
               array[L6] = t2 + t3;
               array[L7] = t1 - t4;
               array[L8] = t2 - t3;
               t1 = array[L1] + t6;
               t2 = array[L1] - t6;
               t3 = t8 - t0;
               t4 = t5 + t7;
               array[L1] = t1 + t4;
               array[L2] = t2 + t3;
               array[L3] = t1 - t4;
               array[L4] = t2 - t3;
            }
         }
      }
   }
```

// fitlpc.c

```
/*******************************************************/
/***     Linear Prediction (LPC) analysis code for     ***/
/***     SIGGRAPH 1998 audio compression cours         ***/
/***         by Perry R  Cook, Princeton University    ***/
/***     This program takes a 16 bit linear signed     ***/
/***     headerless file and compresses it 18:1        ***/
/***     Parameters assumes 8K sampling rate, for      ***/
/***     other sampling rates, adjust block, hop,      ***/
/***     and possibly order accordingly                ***/
/***     Use lpcresyn c to decompress files            ***/
/*******************************************************/

#include <stdio h>
#include <fcntl h>
#include <string h>
#include <math h>

#define MAX_BLOCK 1024
#define MAX_ORDER 30

main(ac,av)
    short ac;
    char *av[];
{
    extern float autocorr(short size,float *data,float *result);
    extern short minvert(short size,float mat[][MAX_ORDER]);
    extern float lpc_from_data(short order, short size, float *data,
float *coeffs);
    extern float predict(short order,short hop_size,float *data,float
*coeffs);
    short short_data[MAX_BLOCK];
    float data[MAX_BLOCK]={0 0},pred_coeffs[MAX_ORDER],
    short out_coeffs[MAX_ORDER];
    FILE *filein,*fileout;
    short n_read;
    short i,j,block_size,hop_size,order;
    long time = 0;
    float pitch,power,temp;

    if (ac==3)  {
      order = 10;
      block_size = 512;
      hop_size = 256;
      filein = fopen(av[1],"rb"),
      if (filein)     {
         fileout = fopen(av[2],"wb");
         fwrite(&order,2,1,fileout),
         fwrite(&hop_size,2,1,fileout);
           n_read = fread(short_data,2,block_size,filein);
         for (i=0;i<block_size,i++) data[i] = short_data[i];
         while (n_read)     {
             pitch = lpc_from_data(order+1,block_size,data,pred_coeffs),
             power = predict(order,hop_size,data,pred_coeffs);
             time += hop_size;
               for (i=0;i<block_size-hop_size;i++)
                   data[i] = data[hop_size+i];
             n_read = fread(short_data,2,hop_size,filein);
             for (i=hop_size;i<block_size;i++)
                   data[i] = short_data[i-hop_size];
             fwrite(&pitch,4,1,fileout);
             fwrite(&power,4,1,fileout);
               for (j=0,j<order;j++)   {
                   temp = 32767 0 * pred_coeffs[j] / (float) order;
                   out_coeffs[j] = temp;
               }
               fwrite(&out_coeffs,2,order,fileout);
         }
```

```
        fclose(filein);
        fclose(fileout);
      }
    else
        printf("I couldn't find your input file!!!\n");
  }
  else
    printf("Format is fitlpc infile outfile lpc\n"),
  return;
}

/***********  This forms the autocorrelation function  **********/
/***********  Also does pitch detection (not very well)  ********/

float autocorr(short size,float *data,float *result)
{
    long i,j,k,
    float temp,norm;

    for (i=0;i<size/2;i++)      {
      result[i] = 0 0;
      for (j=0;j<size-i-1;j++)        {
          result[i] += data[i+j] * data[j];
      }
    }
    temp = result[0];
    j = size*0 04,
    while (result[j]<temp && j < size/2)        {
      temp = result[j];
      j += 1;
    }
    temp = 0 0;
    for (i=j;i<size*0 5;i++) {
      if (result[i]>temp) {
          j = i;
          temp = result[i];
      }
    }
    norm = 1 0 / (float) size;
    k = size/2,
    for (i=0;i<size/2;i++) result[i] *=  (float) (k - i) * norm;
    if ((result[j] / result[0]) < 0 4) j = 0;
    if (j > size/4) j = 0;
    return (float) j;
}

/********  Brute Force Gauss-Jordan Matrix Inversion  **********/

short minvert(short size,float mat[][MAX_ORDER])
{
    short item,row,col,rank,t2;
    float temp,res[MAX_ORDER][MAX_ORDER];
    short ok,zerorow;

    for (row=1;row<=size;row++)     {
      for (col=1;col<=size;col++)     {
          //    fprintf(stdout," %f ",mat[row][col]);
          if (row==col)
            res[row][col] = 1 0;
          else
            res[row][col] = 0 0;
      }
      //    fprintf(stdout,"\n");
    }
    for (item=1;item<=size;item++) {
      if (mat[item][item]==0)         {
          for (row=item;row<=size,row++)   {
            for (col=1;col<=size;col++)     {
                mat[item][col] = mat[item][col] + mat[row][col],
                res[item][col] = res[item][col] + res[row][col];
            }
```

```
        }
      }
      for (row=item;row<=size;row++)  {
          temp=mat[row][item];
          if (temp!=0)        {
            for (col=1;col<=size;col++)     {
                mat[row][col] = mat[row][col] / temp;
                res[row][col] = res[row][col] / temp;
            }
          }
      }
      if (item!=size) {
          for (row=item+1;row<=size;row++)    {
            temp=mat[row][item];
            if (temp!=0)    {
                for (col=1;col<=size;col++) {
                  mat[row][col] = mat[row][col] - mat[item][col];
                  res[row][col] = res[row][col] - res[item][col],
                }
            }
          }
      }
    }
    for (item=2;item<=size,item++)   {
      for (row=1;row<item;row++)      {
          temp = mat[row][item];
          for (col=1,col<=size;col++)    {
            mat[row][col] = mat[row][col] - temp * mat[item][col];
            res[row][col] = res[row][col] - temp * res[item][col];
          }
      }
    }
/*    ok = TRUE,
    rank = 0;
    for (row=1,row<=size;row++) {
      zerorow = TRUE;
      for (col=1;col<=size;col++)     {
          if (mat[row][col]!=0) zerorow = FALSE;
          t2 = (mat[row][col] + 0 5);
          if (row==col&&t2!=1) ok = FALSE;
          t2 = fabs(mat[row][col]*100 0);
          if (row!=col&&t2!=0) ok = FALSE;
      }
      if (!zerorow) rank += 1;
    }
    if (!ok)    {
      fprintf(stdout,"Matrix Not Invertible\n");
      fprintf(stdout,"Rank is Only %i of %i\n",rank,size);
    }
*/
    for (row=1,row<=size;row++) {
      for (col=1;col<=size;col++)     {
          mat[row][col] = res[row][col],
      }
    }
    return rank;
}

/***********  Predictor Coeffs from a block of data   **********/

float lpc_from_data(short order, short size, float *data, float *coeffs)
{
    float r_mat[MAX_ORDER][MAX_ORDER];
    short i,j;
    float pitch;
    float corr[MAX_BLOCK];

    pitch = autocorr(size, data,corr);
    for (i=1;i<order;i++) {
      for (j=1;j<order;j++) r_mat[i][j] = corr[abs(i-j)];
    }
```

```
    minvert(order-1,r_mat);
    for (i=0;i<order-1;i++)     {
      coeffs[i] = 0 0;
      for (j=0;j<order-1;j++) {
          coeffs[i] += r_mat[i+1][j+1] * corr[1+j];
      }
    }
    return pitch;
}

/***********  Test the prediction function           **********/

float predict(short order,short hop_size,float *data,float *coeffs)
{
    short i,j,n_write;
    double power=0 0;
    float error,tmp;
    float Zs[MAX_ORDER];
    for (i=0;i<order;i++) Zs[i] = data[order-i-1];
    for (i=order;i<=hop_size+order;i++) {
      tmp = 0 0;
      for (j=0,j<order;j++)  tmp += Zs[j]*coeffs[j];
      for (j=order-1;j>0;j--) Zs[j] = Zs[j-1];
      Zs[0] = data[i],
      error = data[i] - tmp;
      power += error * error,
    }
    error = sqrt(power);
    error = error / (float) hop_size;
    return error;
}
```

// lpcresyn.c

```
/*******************************************************/
/***    Linear Prediction (LPC) resynthesis code     ***/
/***    for SIGGRAPH 1998 audio compression course   ***/
/***        by Perry R  Cook, Princeton University   ***/
/***    This program takes an LPC compressed file    ***/
/***    and decompresses it to 16 bit linear         ***/
/***    Use fitlpc c to compress files               ***/
/*******************************************************/

#include <stdio h>
#include <stdlib h>
#include <fcntl h>
#include <string h>
#include <math h>

#define MAX_BLOCK 1024
#define MAX_ORDER 30
#define ONE_OVER_RANDLIMIT 0 00003052

main(ac,av)
    short ac;
    char *av[];
{
    float coeffs[MAX_ORDER],Zs[MAX_ORDER],output,input,last_input = 1;
    short in_coeffs[MAX_ORDER];
    FILE *filein,*fileout;
    short n_read;
    short shortout;
    long i,j;
    int hop_size,order,ticker;
    long time = 0;
    float last_pitch = 100, pitch=100;
    float temp;

    if (ac==3)  {
      filein = fopen(av[1],"rb");
      if (filein)     {
         fileout = fopen(av[2],"wb");
         n_read = fread(&order,1,2,filein);
         for (i=0;i<order;i++) Zs[i] = 0 0;
         n_read = fread(&hop_size,2,1,filein);
         n_read = fread(&pitch,4,1,filein);
         n_read = fread(&input,4,1,filein);
           n_read = fread(in_coeffs,2,order,filein);
           ticker = last_pitch;
           while (n_read)     {
             for (i=0;i<hop_size;i++) {
                output = 0 0;
                    if (input < last_input)
                        last_input *= 0 99;
                    if (input > last_input)
                        last_input *= 1 01;
                if (pitch==0)                       //  Noise Source Input
                        output = last_input * 40 0 *
                        ((random(32768) - 16384) * ONE_OVER_RANDLIMIT),
                else {                              //  Pulse Train Input
                  ticker -= 1;
```

```
                if (ticker <= 0) {
                        ticker = last_pitch;
                        output = last_input * last_pitch * -3;
                }
                    if (pitch < last_pitch)
                        last_pitch *= 0 995;
                    if (pitch > last_pitch)
                        last_pitch *= 1 005;
                  }
              for (j=0;j<order;j++)     {      //  Get Filter Coeffs
                    temp = (float) in_coeffs[j]  / (float) 32767 0;
                    temp = temp * (float) order;
                    coeffs[j] = temp;
                  }
                  for (j=0;j<order;j++)       //  Do Prediction Filter
                output += Zs[j]*coeffs[j];
              for (j=order-1,j>0;j--) Zs[j] = Zs[j-1];
              Zs[0] = output;
              if (abs(output)>32000) {          //  Check for Overload
                if (output < 0) output = -32000,
                else            output = 32000;
                    for (j=0;j<order;j++) Zs[j] *= 0 9;
              }
              shortout = output;
                  fwrite(&shortout,2,1,fileout);
            }
            time += hop_size;
              n_read = fread(&pitch,4,1,filein);
            n_read = fread(&input,4,1,filein),
              n_read = fread(in_coeffs,2,order,filein);
        }
        fclose(filein),
        fclose(fileout);
      }
    else
        printf("I couldn't find your input file!!!\n");
  }
  else
    printf("Format is lpcresyn infile lpc outfile raw\n");
  return;
}
```

// sines.c

```
/*****************************************************/
/***    Program to make sine waves of various      ***/
/***    amplitudes and frequencies                 ***/
/***    For SIGGRAPH 1998 audio compression        ***/
/***    course, by Perry R  Cook, Princeton U      ***/
/*****************************************************/

#include <math h>
#include <stdio h>

main()  {
    long i,j;
    short data;
    FILE *fileOut;
    float amp;

    fileOut = fopen("sines raw","wb");
    for (j=15,j>0;j--)  {
        amp = pow(2 0,(float) j);
        for (i=0,i<4096,i++)    {
            data = amp * sin(0 04 * i);
            fwrite(&data,2,1,fileOut),
        }
        for (i=0,i<4096;i++)    {
            data = amp * sin(0 16 * i),
            fwrite(&data,2,1,fileOut),
        }
        for (i=0;i<4096;i++)    {
            data = amp * sin(0 64 * i);
            fwrite(&data,2,1,fileOut);
        }
    }
    fclose(fileOut);
}
```

// byteswap.c

```
#include <stdlib h>
#include <stdio h>
#include <string h>

#define INT16 short
#define INT32 long

/*************************************************************/
/***   BYTE SWAP FOR MOTOROLA/INTEL COMPATABILITY   *****/
/*************************************************************/

INT16 byte_swap(INT16 number)
{
    INT16 *temp,
    char swapper[3];
    temp = (INT16 *) swapper,
    *temp = number;
    swapper[2] = swapper[0];
    swapper[0] = swapper[1],
    swapper[1] = swapper[2];
    return *temp;
}

void main(int argc, char *argv[]) {
    FILE *file_in,*file_out;
    INT16  data[512],
    INT16 i,j;
    if (argc!=3)        {
      printf("useage: byte_swap infile outfile\n");
      exit(0),
    }
    file_in = fopen(argv[1],"rb");
    if (file_in)   {
      file_out = fopen(argv[2],"wb"),
      j = fread(data,2,512,file_in);
      while (j>0) {
          for (i=0;i<j;i++)
            data[i] = byte_swap(data[i]);
          fwrite(data,2,j,file_out);
          j = fread(data,2,512,file_in);
      }
    }
    fclose(file_in);
    fclose(file_out);
    exit(0);
}
```

// diffrnce.c

```
/*********************************************************/
/***    Program computes difference between two        ***/
/***    raw headerless files   Good for determining    ***/
/***    errors   Won't necessarily work for all        ***/
/***    compression algorithms (LPC for example)       ***/
/***    for SIGGRAPH 1998 compression course           ***/
/***        by Perry R  Cook, Princeton University     ***/
/*********************************************************/

#include <stdio h>

main(int argc, char *argv[])  {
    FILE *fileIn1, *fileIn2, *fileOut;
    short data1,data2;

    fileIn1 = fopen(argv[1],"rb");
    fileIn2 = fopen(argv[2],"rb");
    fileOut = fopen(argv[3],"wb");
    if (!fileIn1 || !fileIn2 || !fileOut)       {
        printf("useage  %s in1 raw in2 raw out raw\n",argv[0]);
        fclose(fileIn1);
        fclose(fileIn2);
        fclose(fileOut),
    }
    while (fread(&data1,2,1,fileIn1) && fread(&data2,2,1,fileIn2))
{
        data1 = data1 - data2,
        fwrite(&data1,2,1,fileOut),
    }
    fclose(fileIn1);
    fclose(fileIn2);
    fclose(fileOut),
}
```